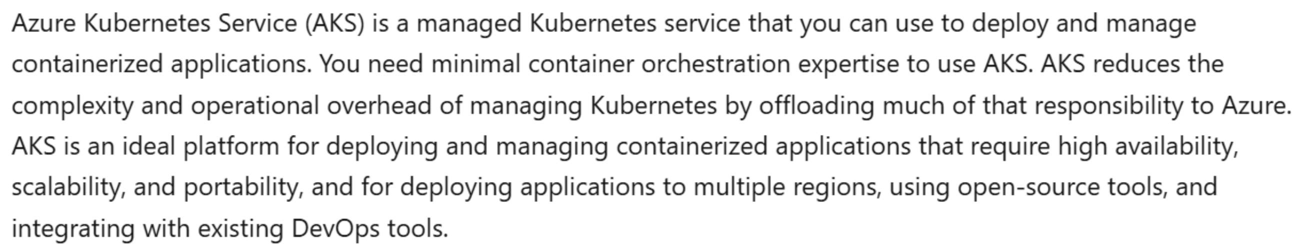
Exhibit 2

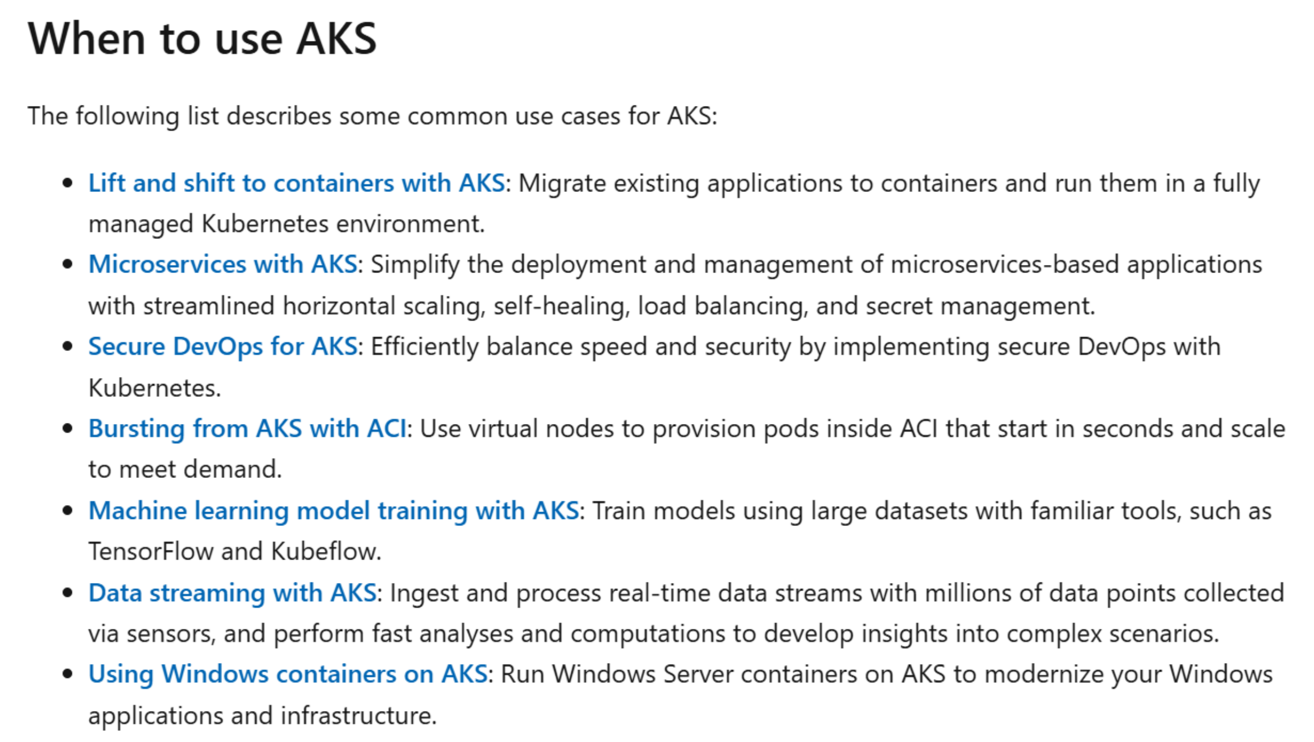
Exhibit 4

U.S. Patent No. 7,784,058 vs. Microsoft

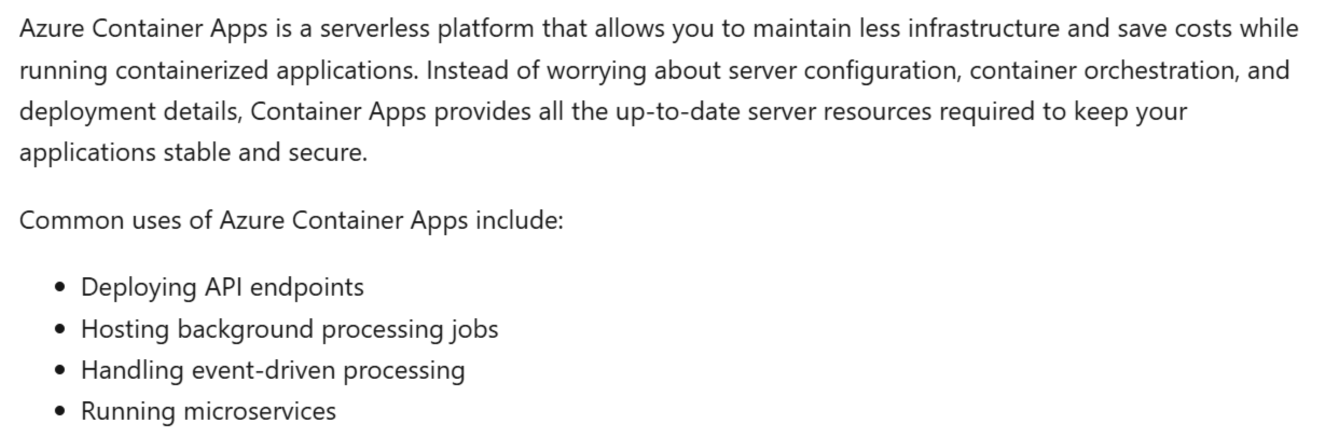
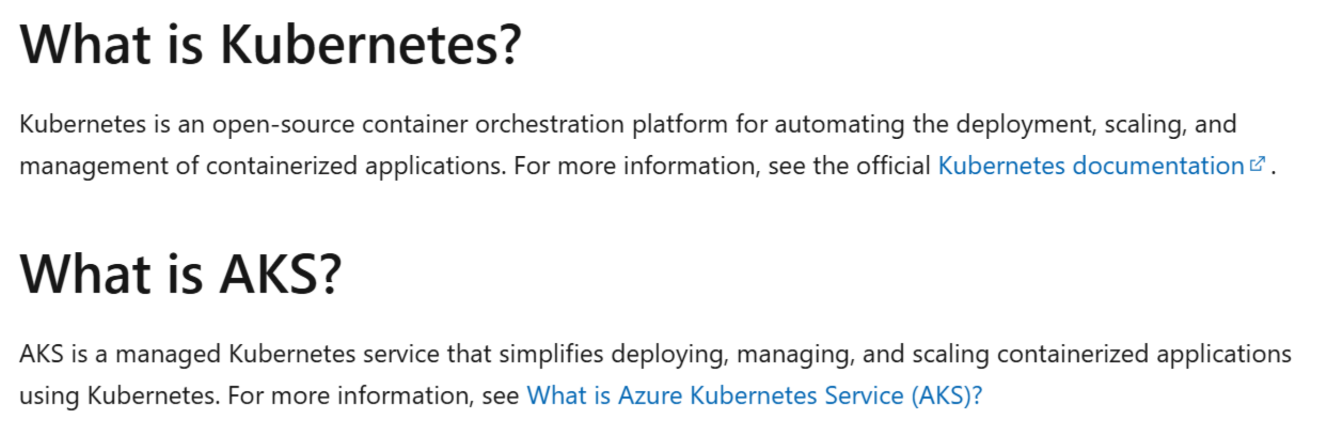
Accused Instrumentalities: Microsoft products and services using user mode critical system elements as shared libraries, including without limitation Azure Kubernetes Service ("AKS"), Azure Arc-enabled Kubernetes, Azure Container Registry, and Azure Container Apps, and all versions and variations thereof since the issuance of the asserted patent.

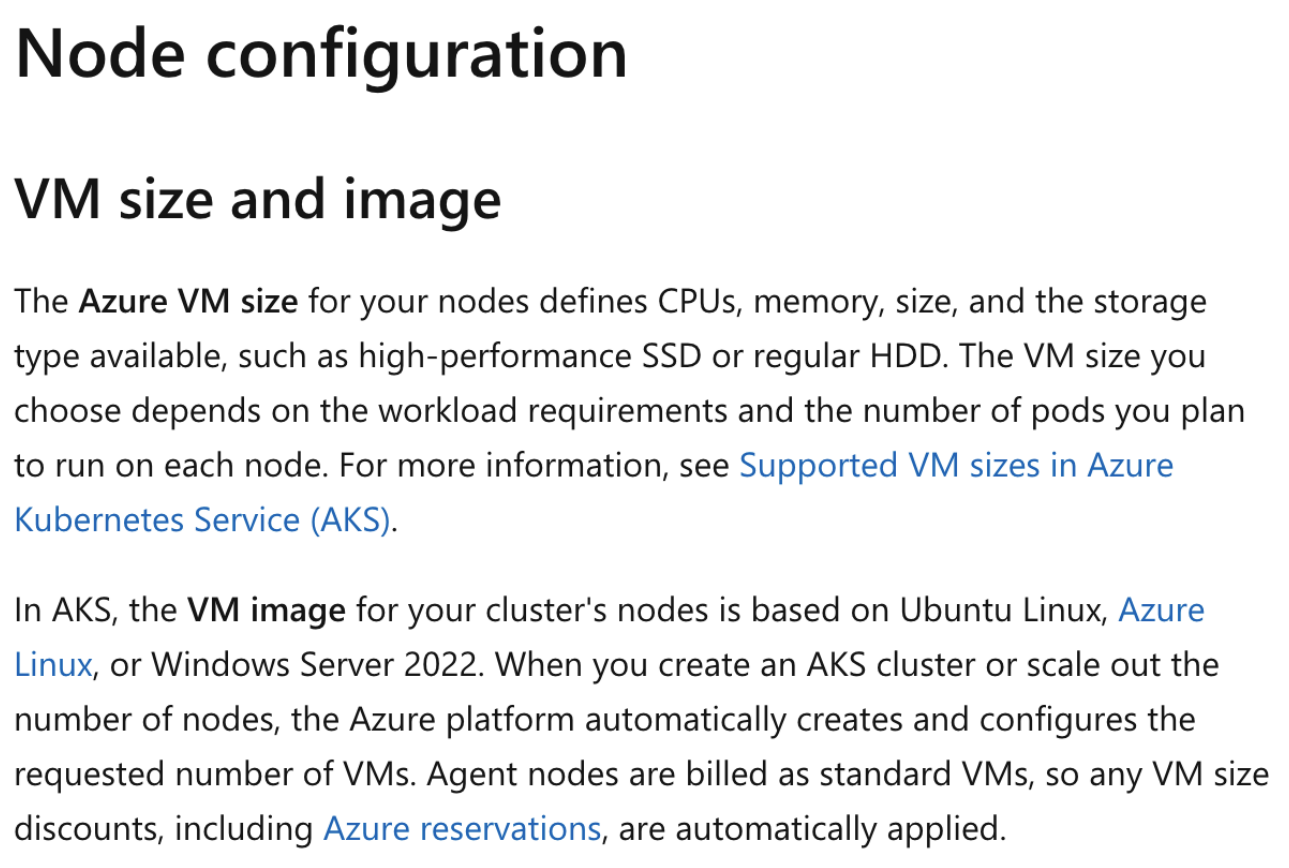
Claim 1

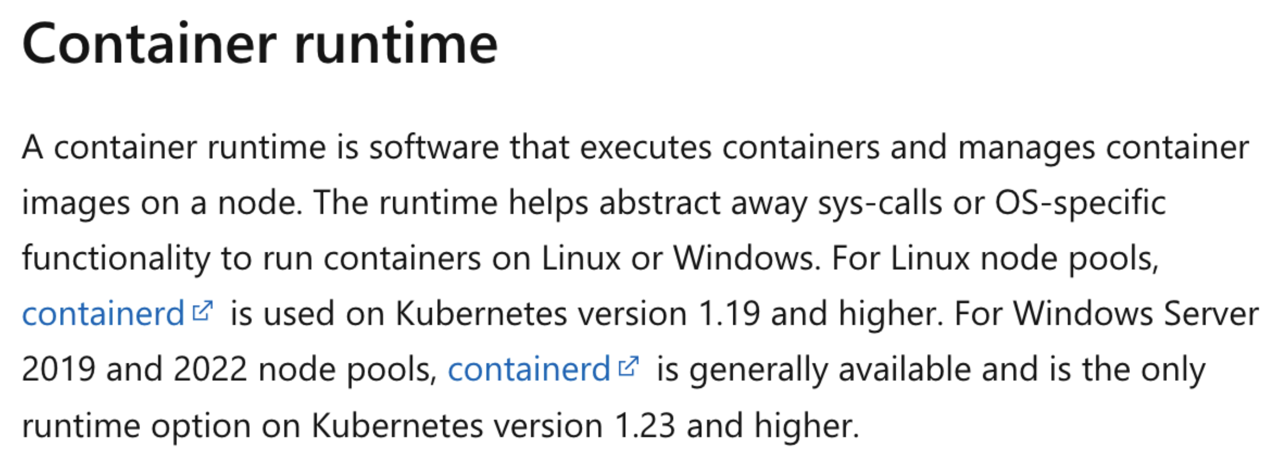
Claim 1	Accused Instrumentalities
[1pre] 1. A computing system for executing a plurality of software applications comprising:	To the extent the preamble is limiting, each Accused Instrumentality comprises or constitutes a computing system for executing a plurality of software applications as claimed. *See* claim limitations below. *See also, e.g.*:  Azure Kubernetes Service (AKS) is a managed Kubernetes service that you can use to deploy and manage containerized applications. You need minimal container orchestration expertise to use AKS. AKS reduces the complexity and operational overhead of managing Kubernetes by offloading much of that responsibility to Azure. AKS is an ideal platform for deploying and managing containerized applications that require high availability, scalability, and portability, and for deploying applications to multiple regions, using open-source tools, and integrating with existing DevOps tools. https://learn.microsoft.com/en-us/azure/aks/what-is-aks

Claim 1	Accused Instrumentalities
	 **When to use AKS** The following list describes some common use cases for AKS: • Lift and shift to containers with AKS: Migrate existing applications to containers and run them in a fully managed Kubernetes environment. • Microservices with AKS: Simplify the deployment and management of microservices-based applications with streamlined horizontal scaling, self-healing, load balancing, and secret management. • Secure DevOps for AKS: Efficiently balance speed and security by implementing secure DevOps with Kubernetes. • Bursting from AKS with ACI: Use virtual nodes to provision pods inside ACI that start in seconds and scale to meet demand. • Machine learning model training with AKS: Train models using large datasets with familiar tools, such as TensorFlow and Kubeflow. • Data streaming with AKS: Ingest and process real-time data streams with millions of data points collected via sensors, and perform fast analyses and computations to develop insights into complex scenarios. • Using Windows containers on AKS: Run Windows Server containers on AKS to modernize your Windows applications and infrastructure. https://learn.microsoft.com/en-us/azure/aks/what-is-aks

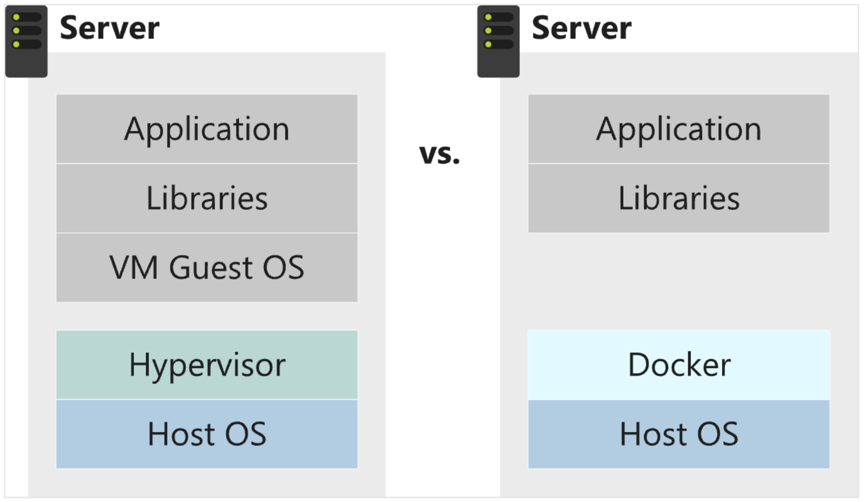
Claim 1	Accused Instrumentalities
	Azure Arc-enabled Kubernetes allows you to attach Kubernetes clusters running anywhere so that you can manage and configure them in Azure. By managing all of your Kubernetes resources in a single control plane, you can enable a more consistent development and operation experience, helping you run cloud-native apps anywhere and on any Kubernetes platform. When the Azure Arc agents are deployed to the cluster, an outbound connection to Azure is initiated, using industry-standard SSL to secure data in transit. Clusters that you connect to Azure are represented as their own resources in Azure Resource Manager, and they can be organized using resource groups and tagging. https://learn.microsoft.com/en-us/azure/azure-arc/kubernetes/overview Containers are becoming the preferred way to package, deploy, and manage cloud applications. Azure Container Instances offers the fastest and simplest way to run Linux or Windows containers in Azure, without having to manage any virtual machines and without having to adopt a higher-level service. ACI supports regular, confidential, and Spot containers. ACI can be used as single-instance or multi-instance via NGroups, or you can get orchestration capabilities by deploying pods in your Azure Kubernetes Service (AKS) cluster via virtual nodes on ACI. For even faster startup times, ACI supports standby pools. https://learn.microsoft.com/en-us/azure/container-instances/container-instances-overview

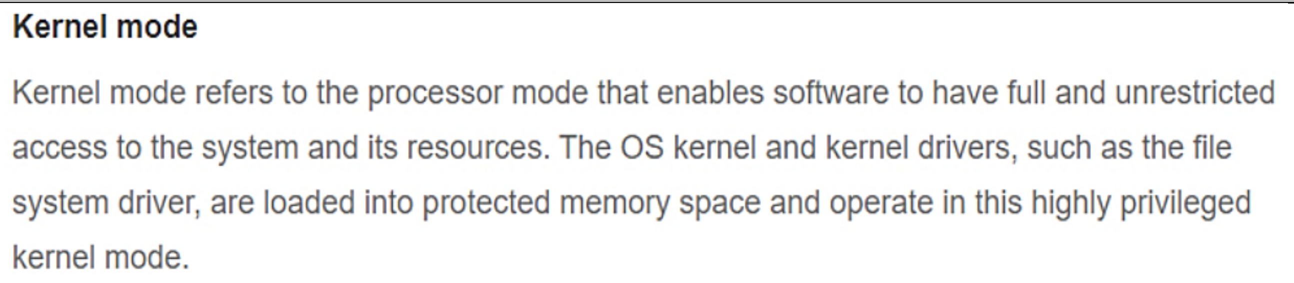
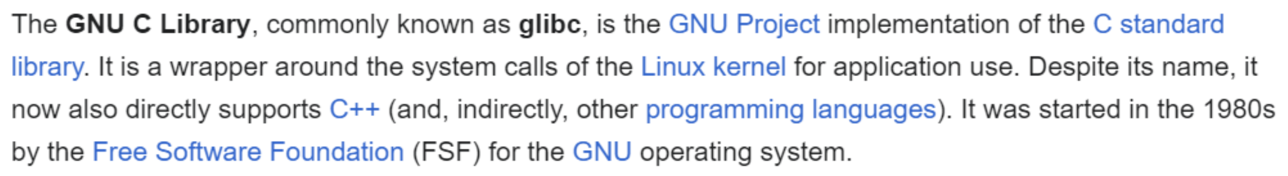
Claim 1	Accused Instrumentalities
	 Azure Container Apps is a serverless platform that allows you to maintain less infrastructure and save costs while running containerized applications. Instead of worrying about server configuration, container orchestration, and deployment details, Container Apps provides all the up-to-date server resources required to keep your applications stable and secure. Common uses of Azure Container Apps include: • Deploying API endpoints • Hosting background processing jobs • Handling event-driven processing • Running microservices https://learn.microsoft.com/en-us/azure/container-apps/overview  **What is Kubernetes?** Kubernetes is an open-source container orchestration platform for automating the deployment, scaling, and management of containerized applications. For more information, see the official Kubernetes documentation . **What is AKS?** AKS is a managed Kubernetes service that simplifies deploying, managing, and scaling containerized applications using Kubernetes. For more information, see What is Azure Kubernetes Service (AKS)? https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts

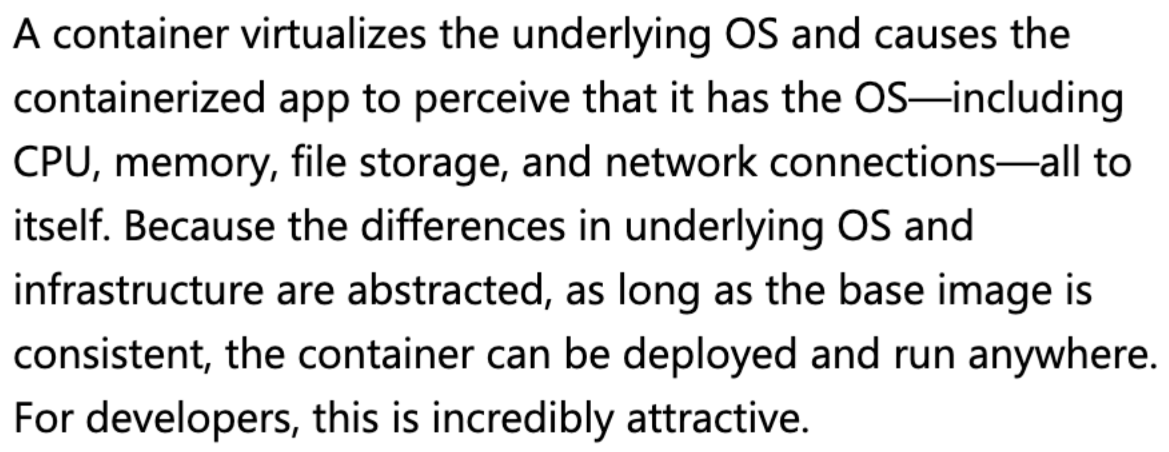
Claim 1	Accused Instrumentalities
[1a] a) a processor;	Each Accused Instrumentality comprises a processor. For example, each node/host contains at least one CPU. *See, e.g.*:  Node configuration VM size and image The **Azure VM size** for your nodes defines CPUs, memory, size, and the storage type available, such as high-performance SSD or regular HDD. The VM size you choose depends on the workload requirements and the number of pods you plan to run on each node. For more information, see Supported VM sizes in Azure Kubernetes Service (AKS). In AKS, the **VM image** for your cluster's nodes is based on Ubuntu Linux, Azure Linux, or Windows Server 2022. When you create an AKS cluster or scale out the number of nodes, the Azure platform automatically creates and configures the requested number of VMs. Agent nodes are billed as standard VMs, so any VM size discounts, including Azure reservations, are automatically applied. https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts

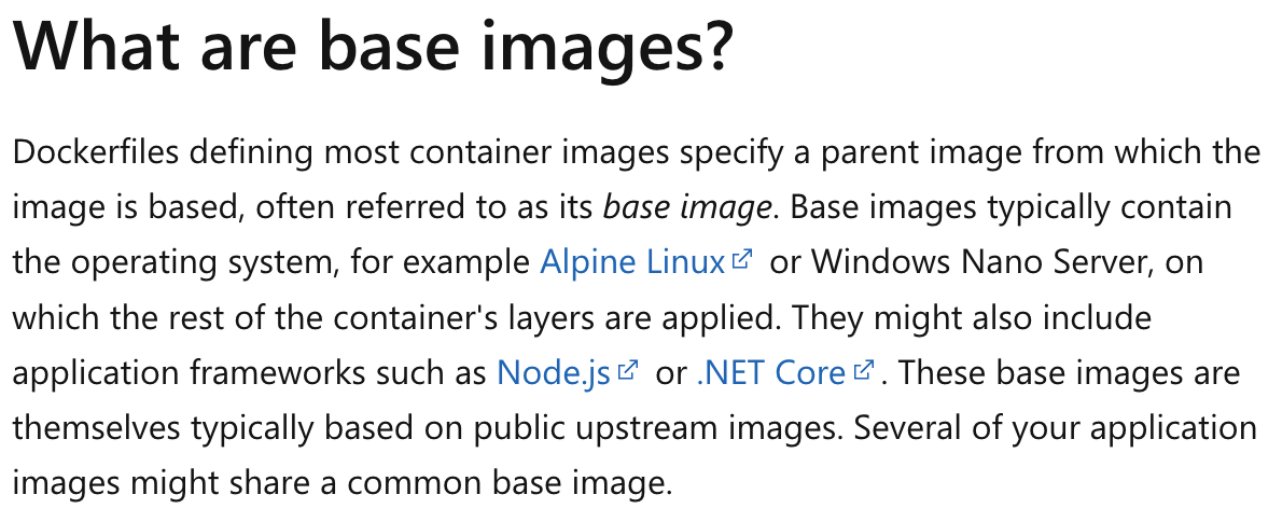
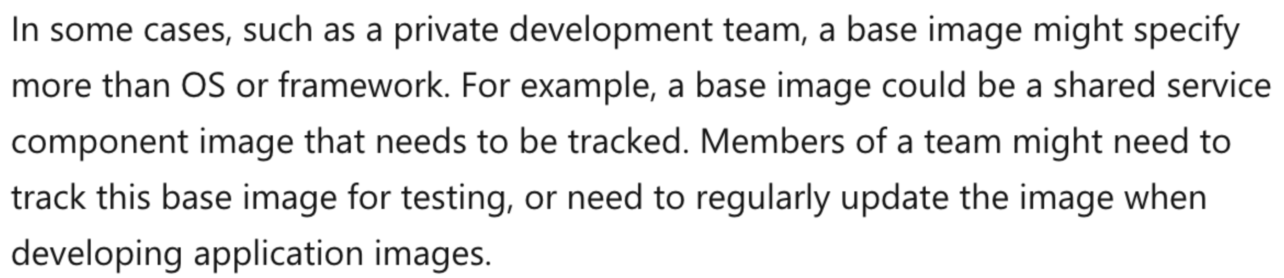
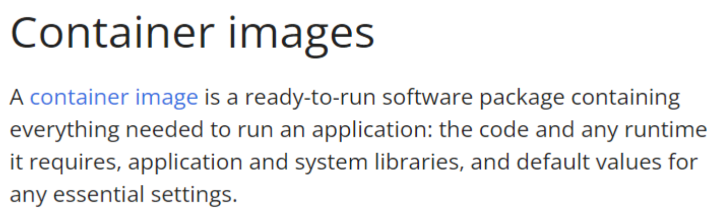
Claim 1	Accused Instrumentalities
[1b] b) an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor; and,	Each Accused Instrumentality comprises an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor. For example, the OSCSEs include kernel-mode functions similar to the functionalities provided by user-space libraries such as glibc. These are implemented in kernel-space to handle tasks such as (without limitation) memory management (kmalloc(), kfree(), etc,) at kernel level. *See, e.g.*:  **Container runtime** A container runtime is software that executes containers and manages container images on a node. The runtime helps abstract away sys-calls or OS-specific functionality to run containers on Linux or Windows. For Linux node pools, containerd is used on Kubernetes version 1.19 and higher. For Windows Server 2019 and 2022 node pools, containerd is generally available and is the only runtime option on Kubernetes version 1.23 and higher. https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts

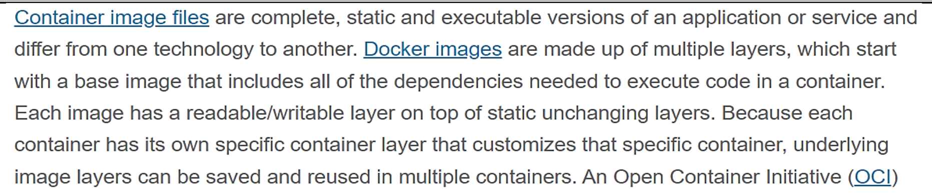
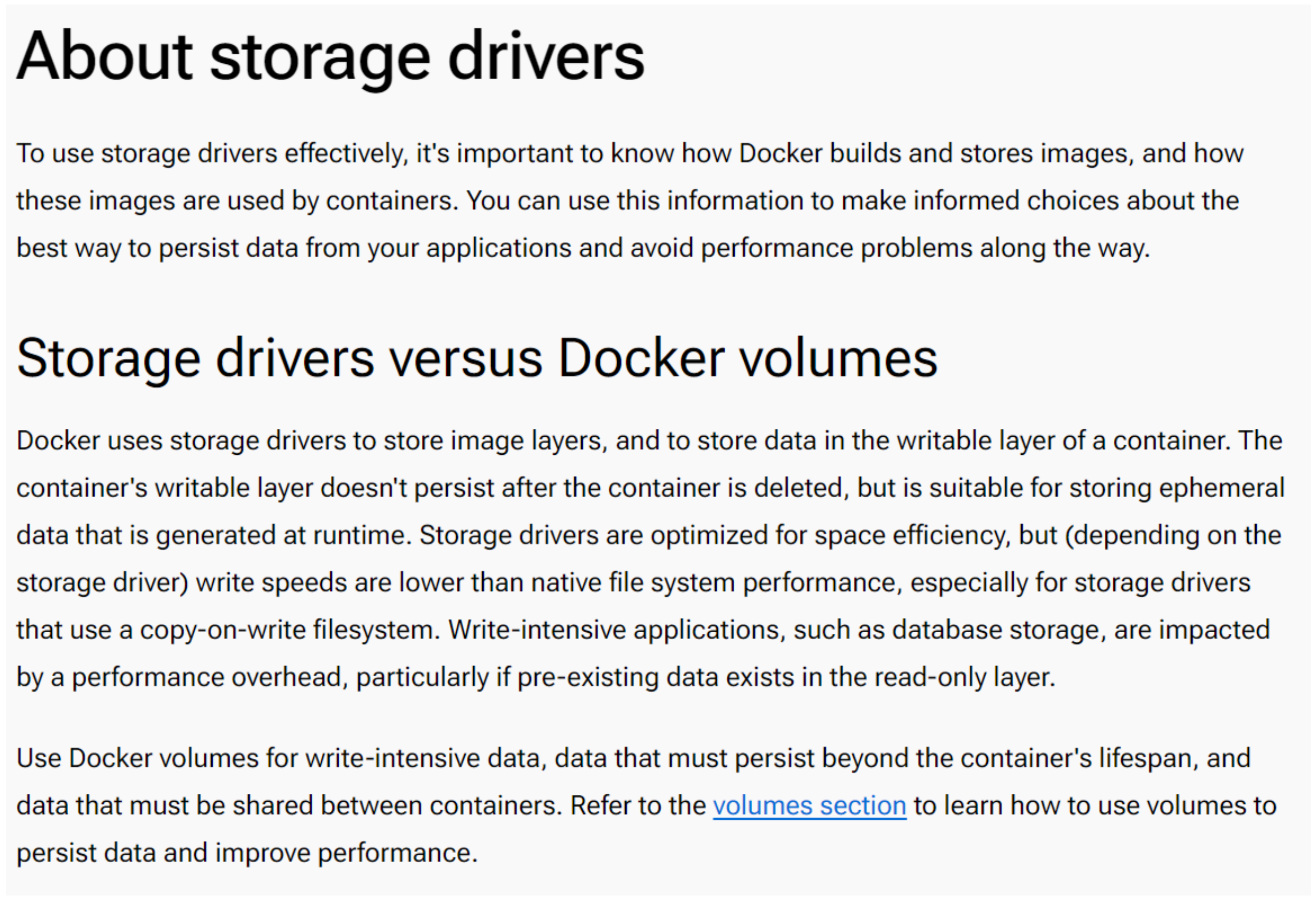
Case 4:24-cv-04740-BB Document 1-6 Filed 12/20/24 Page 8 of 48

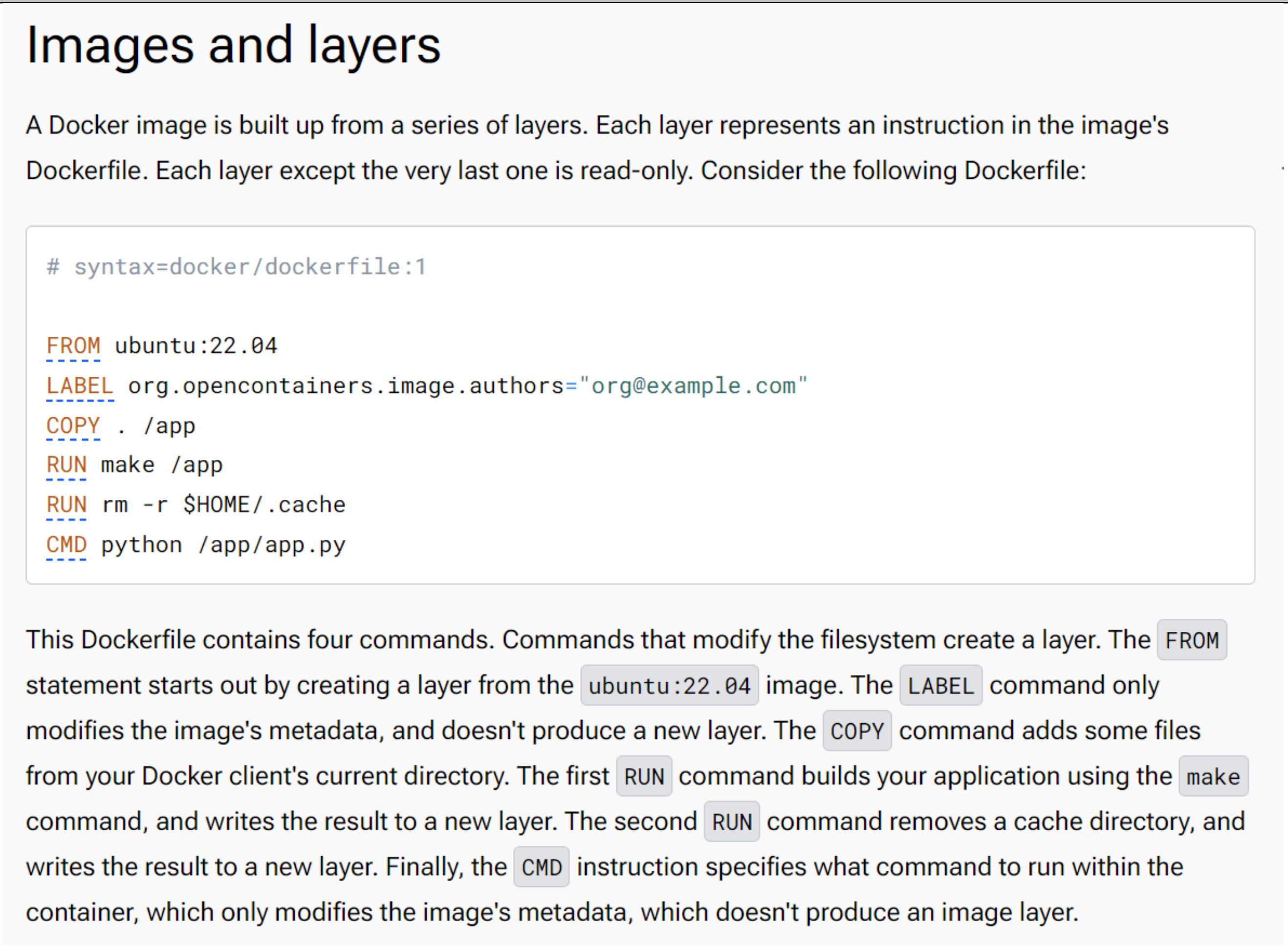
Claim 1	Accused Instrumentalities
	## Docker benefits When we use Docker, we immediately get access to the benefits containerization offer. ### Efficient hardware use Containers run without using a virtual machine (VM). As we learned, the container relies on the host kernel for functions such as file system, network management, process scheduling, and memory management.  Server Application Libraries VM Guest OS Hypervisor Host OS vs. Server Application Libraries Docker Host OS Compared to a VM, we can see that a VM requires an OS installed to provide kernel functions to the running applications inside the VM. Keep in mind that the VM OS also requires disk space, memory, and CPU time. By removing the VM and the additional OS requirement, we can free resources on the host and use it for running other containers. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

Claim 1	Accused Instrumentalities
	 **Kernel mode** Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode. https://www.techtarget.com/searchdatacenter/definition/kernel  The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc

Claim 1	Accused Instrumentalities
[1c] c) a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode and	Each Accused Instrumentality comprises a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode. For example, the shared library with SLCSEs include the runtime environment, system tools, and dependencies, such as the glibc library and other libraries that replicate OSCSEs, included in the container image (including without limitation in a base image that is included within the container image). *See, e.g.*:  A container virtualizes the underlying OS and causes the containerized app to perceive that it has the OS—including CPU, memory, file storage, and network connections—all to itself. Because the differences in underlying OS and infrastructure are abstracted, as long as the base image is consistent, the container can be deployed and run anywhere. For developers, this is incredibly attractive. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-container/

Claim 1	Accused Instrumentalities
	 ## What are base images? Dockerfiles defining most container images specify a parent image from which the image is based, often referred to as its *base image*. Base images typically contain the operating system, for example Alpine Linux or Windows Nano Server, on which the rest of the container's layers are applied. They might also include application frameworks such as Node.js or .NET Core. These base images are themselves typically based on public upstream images. Several of your application images might share a common base image. https://learn.microsoft.com/en-us/azure/container-registry/container-registry-tasks-base-images  In some cases, such as a private development team, a base image might specify more than OS or framework. For example, a base image could be a shared service component image that needs to be tracked. Members of a team might need to track this base image for testing, or need to regularly update the image when developing application images. https://learn.microsoft.com/en-us/azure/container-registry/container-registry-tasks-base-images  ### Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/

Claim 1	Accused Instrumentalities
	 Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI) https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization  **About storage drivers** To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	 https://docs.docker.com/storage/storagedriver/

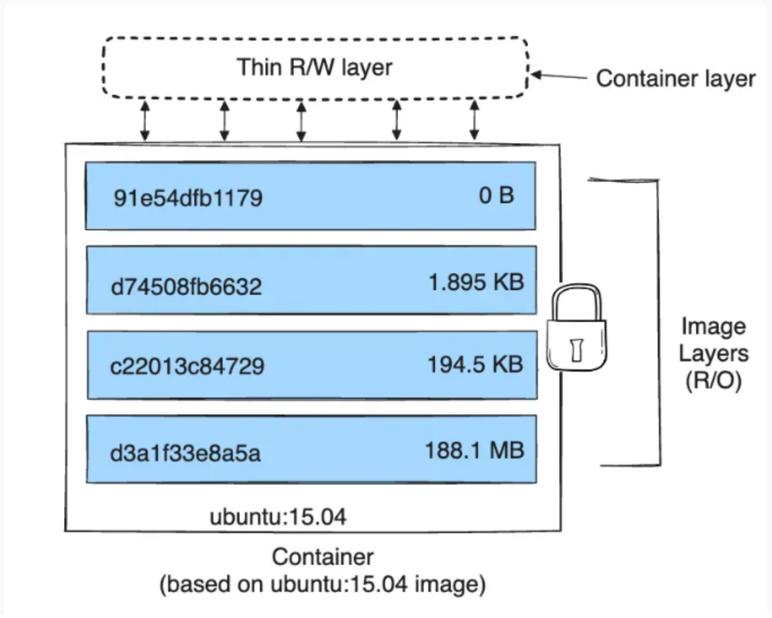
Images and layers

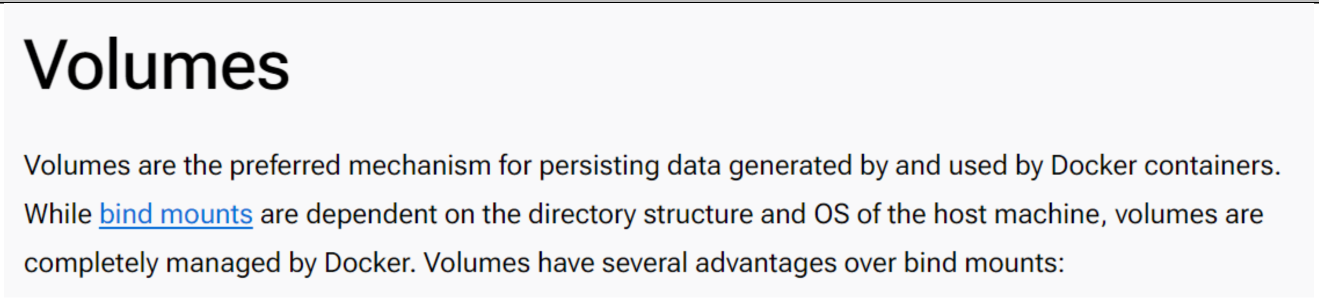
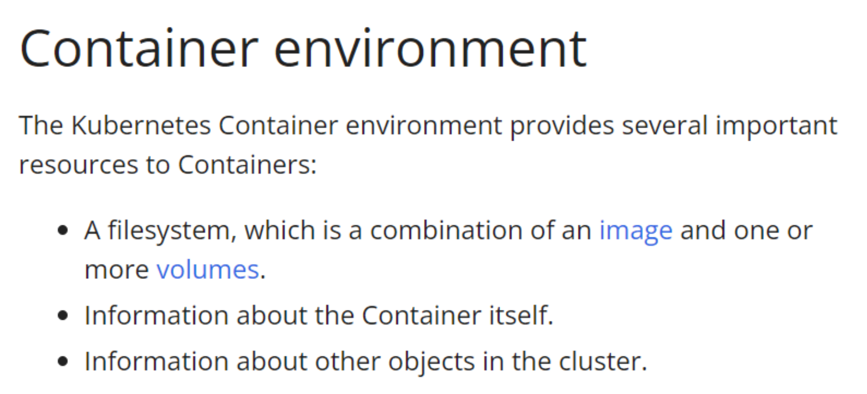
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

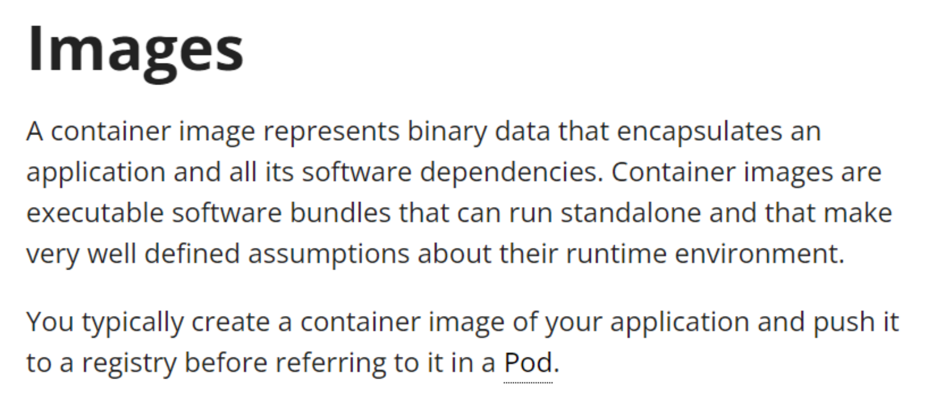
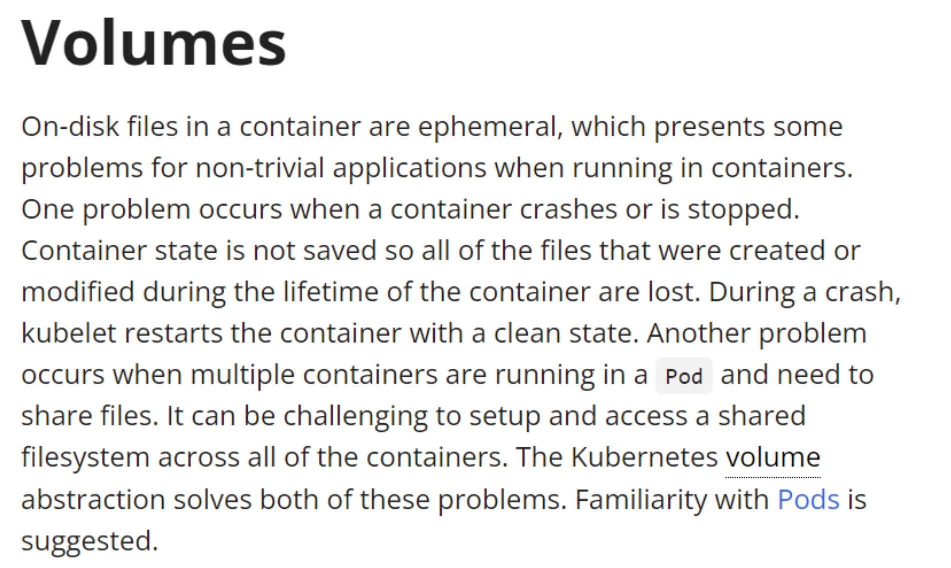
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

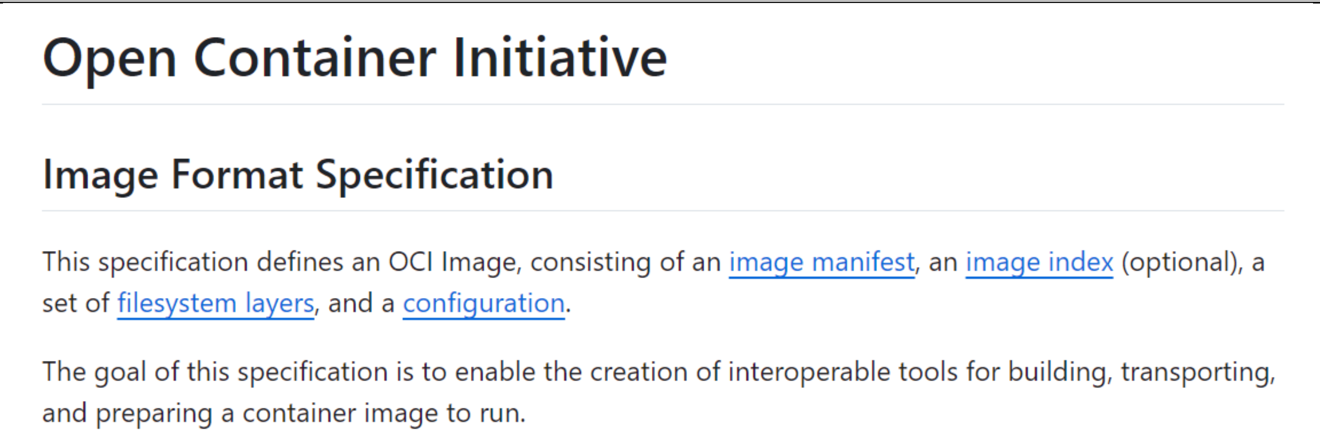
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

Case 5:24-cv-04740-PCP Document 43-3 Filed 12/20/24 Page 14 of 48

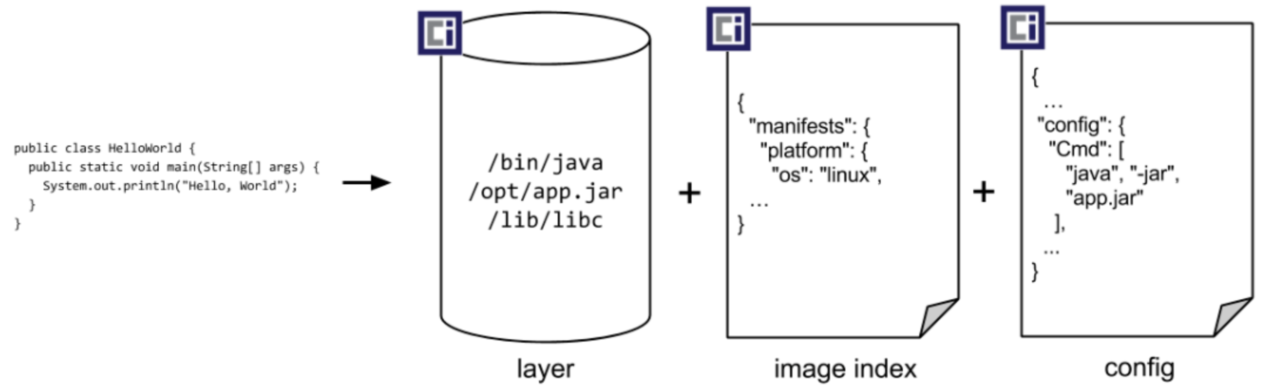
Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB Image Layers (R/O) ubuntu:15.04 Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	 Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/  Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 1	Accused Instrumentalities
	 ## Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  ## Volumes On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Case 4:24-cv-04003-PCP Document 46-3 Filed 02/10/25 Page 18 of 48

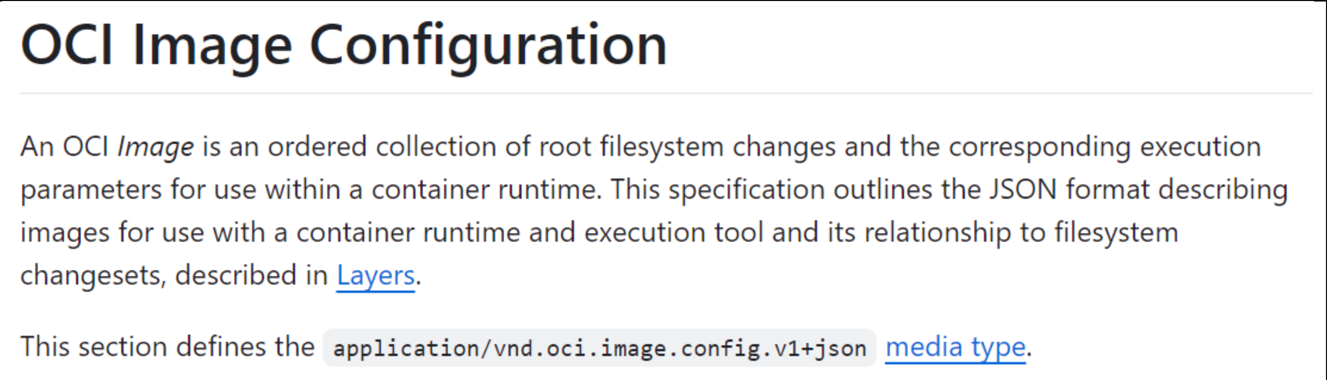
Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

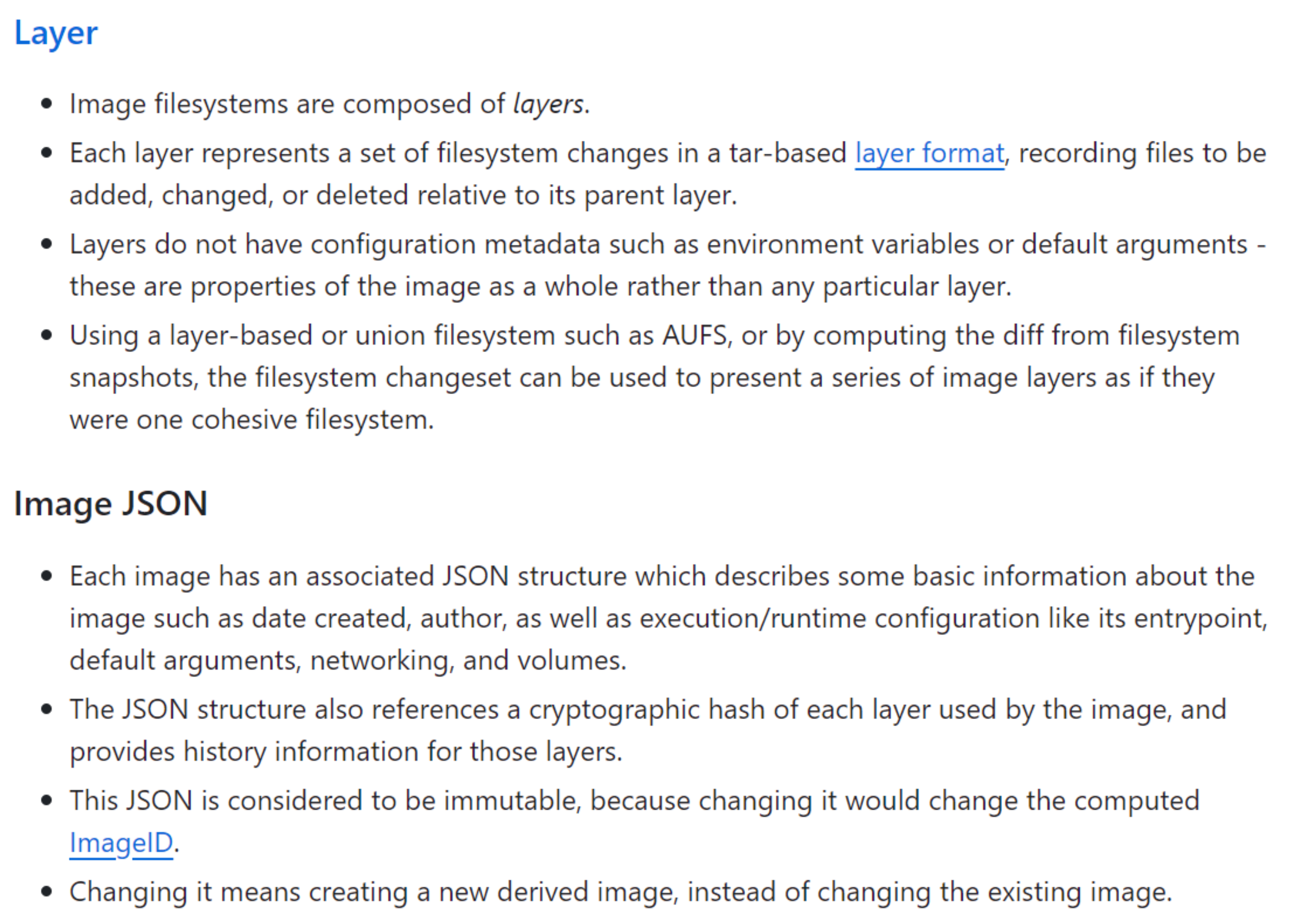
```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World");
  }
}
```

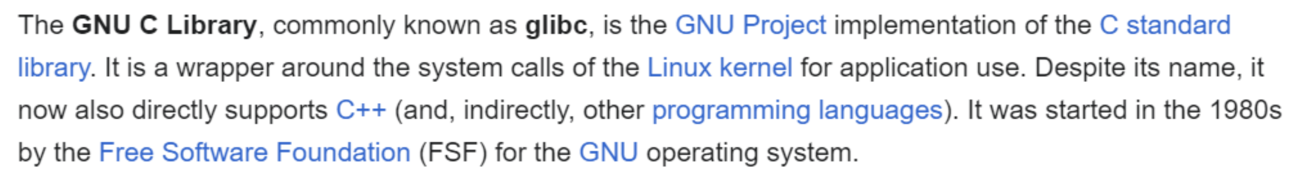
```
/bin/java
/opt/app.jar
/lib/libc
```

```
{
  "manifests": {
    "platform": {
      "os": "linux",
  ...
}
```

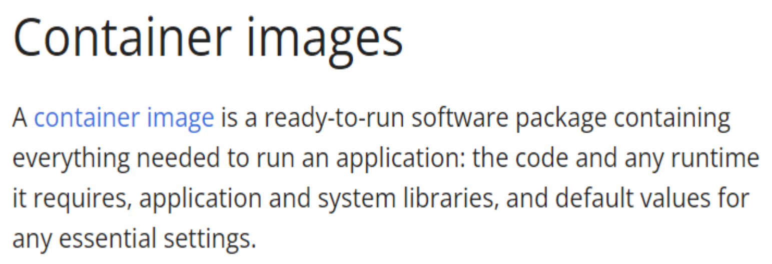
```
{
  ...
  "config": {
    "Cmd": [
      "java", "-jar",
      "app.jar"
    ],
  ...
}
```

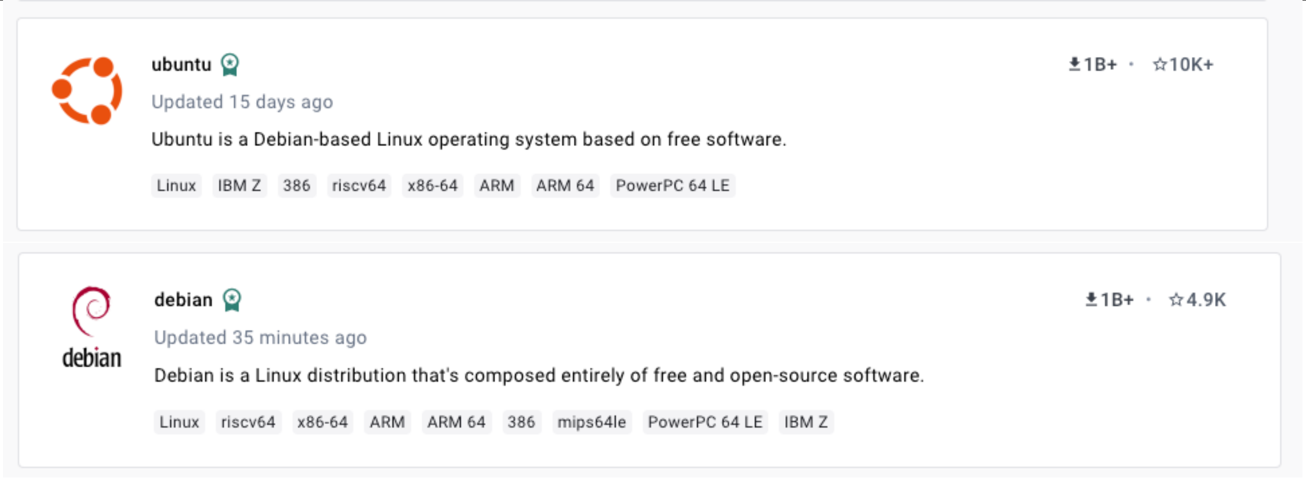
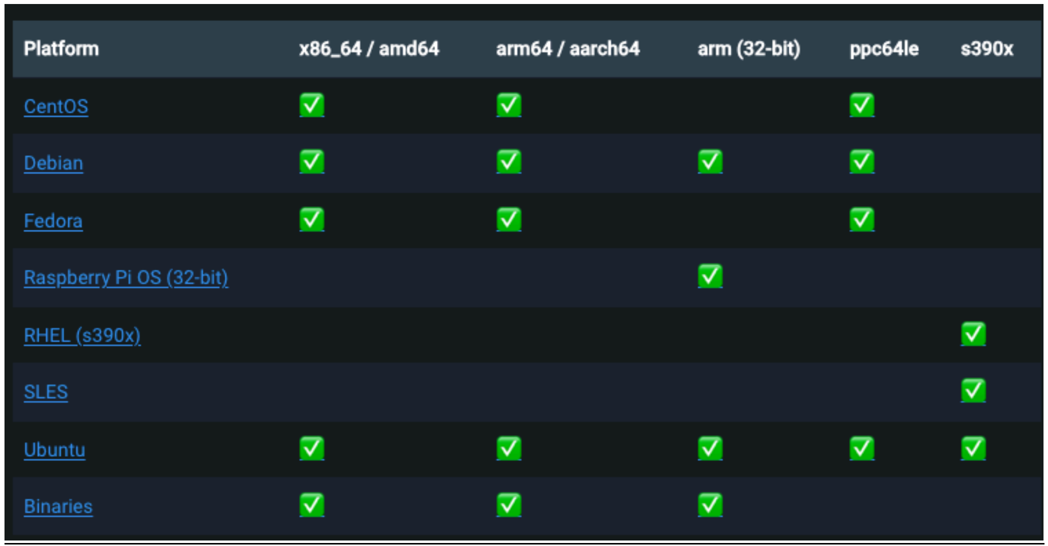
Claim 1	Accused Instrumentalities
	 ## OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

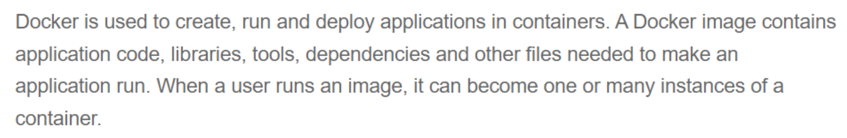
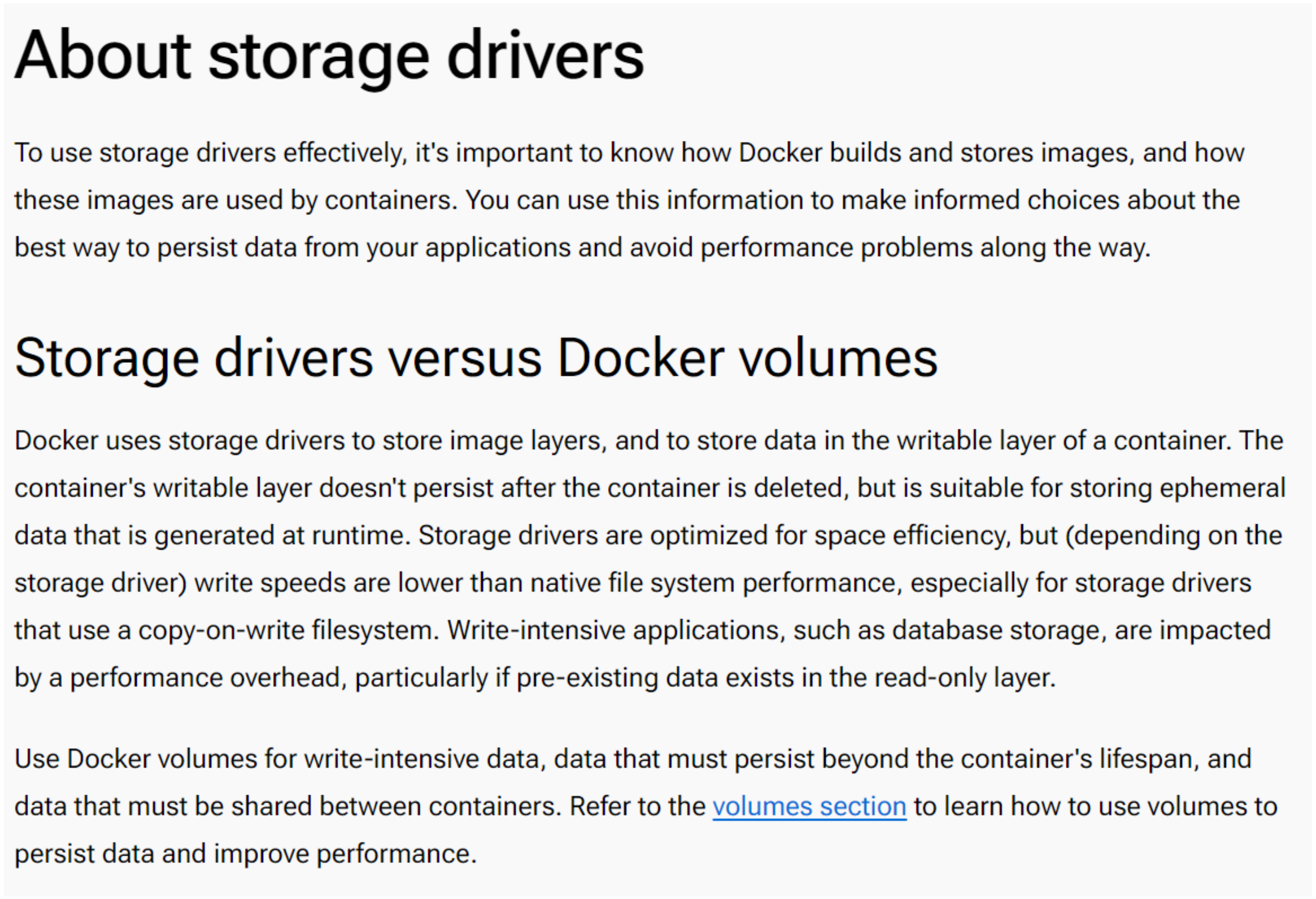
Claim 1	Accused Instrumentalities
	 The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc

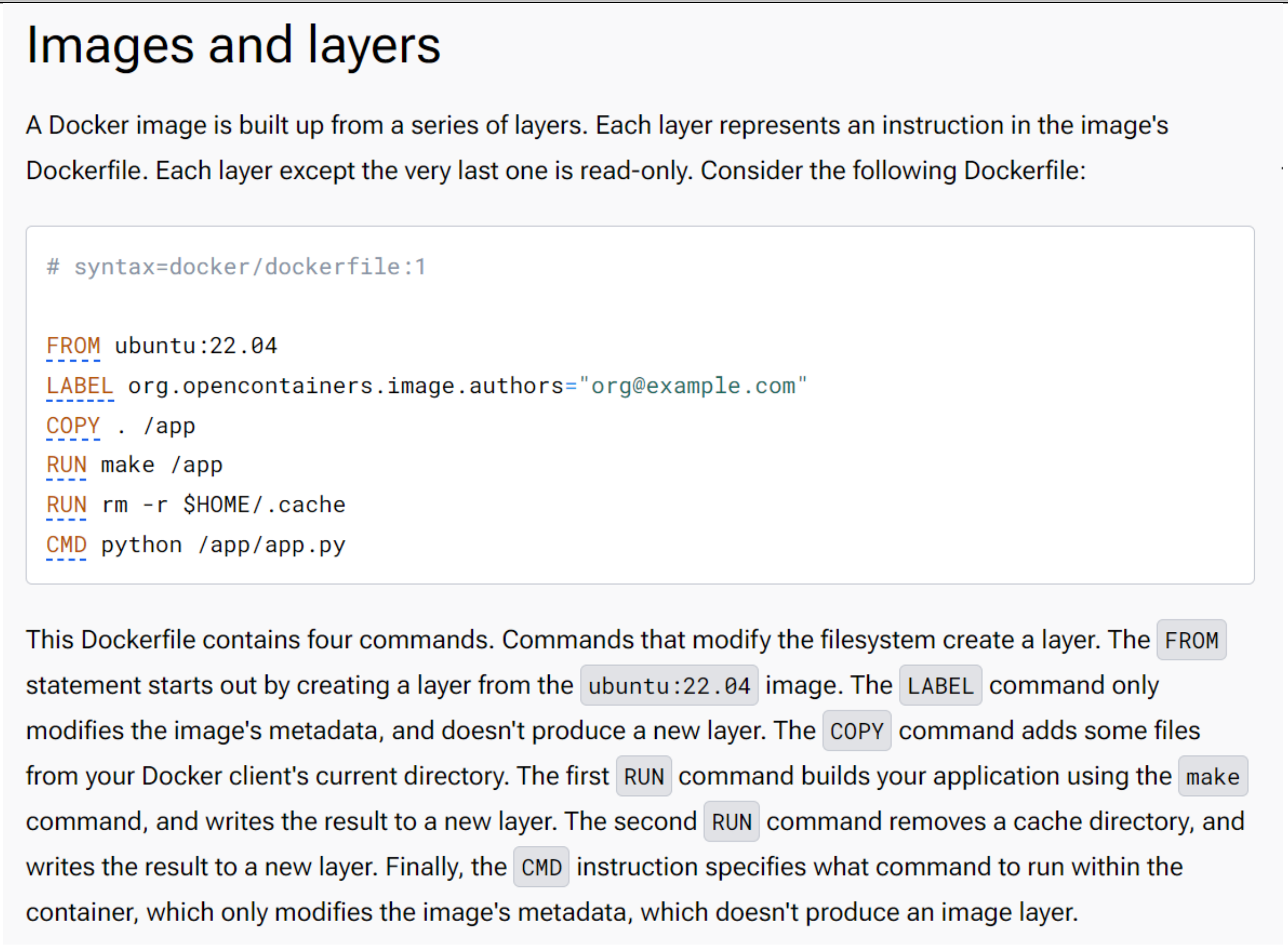
Case 5:24-cv-04740-PCP Document 1-43-3 Filed 12/20/24/25 Page 22 of 48

Claim 1	Accused Instrumentalities
[1d] i) wherein some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications,	In each Accused Instrumentality, some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications. For example, a base image serves as a self-contained unit that encompasses all the necessary components for an application to run, including the application code, runtime environment, system tools, and dependencies (i.e., SLCSEs). The images are based on existing Linux distributions, such as Debian and Ubuntu, including essential system elements (i.e., functional replicas of OSCSEs). Each container image is based on a specific base image, which contains the application code, and dependencies, including system libraries or shared library critical system elements (SLCSEs). The base image forms a part of the container image according to the "layer" model described in the documentation below. When the container runs the image, it creates a runtime instance of that container image. In turn, when one or more applications executes within the container runtime environment, it dynamically links to the SLCSEs stored in the runtime environment, which thereby become a part of the application(s). *See, e.g.*:  Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/

Claim 1	Accused Instrumentalities
	 ubuntu Updated 15 days ago Ubuntu is a Debian-based Linux operating system based on free software. Linux IBM Z 386 riscv64 x86-64 ARM ARM 64 PowerPC 64 LE 1B+ · 10K+ debian Updated 35 minutes ago Debian is a Linux distribution that's composed entirely of free and open-source software. Linux riscv64 x86-64 ARM ARM 64 386 mips64le PowerPC 64 LE IBM Z 1B+ · 4.9K https://hub.docker.com/search?image_filter=official&type=image&q=  https://docs.docker.com/engine/install/

Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	ppc64le	s390x
CentOS	✓	✓		✓	
Debian	✓	✓	✓	✓	
Fedora	✓	✓		✓	
Raspberry Pi OS (32-bit)			✓		
RHEL (s390x)					✓
SLES					✓
Ubuntu	✓	✓	✓	✓	✓
Binaries	✓	✓	✓		

Claim 1	Accused Instrumentalities
	 Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image  About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	 https://docs.storage.docker.com/storage/storagedriver/

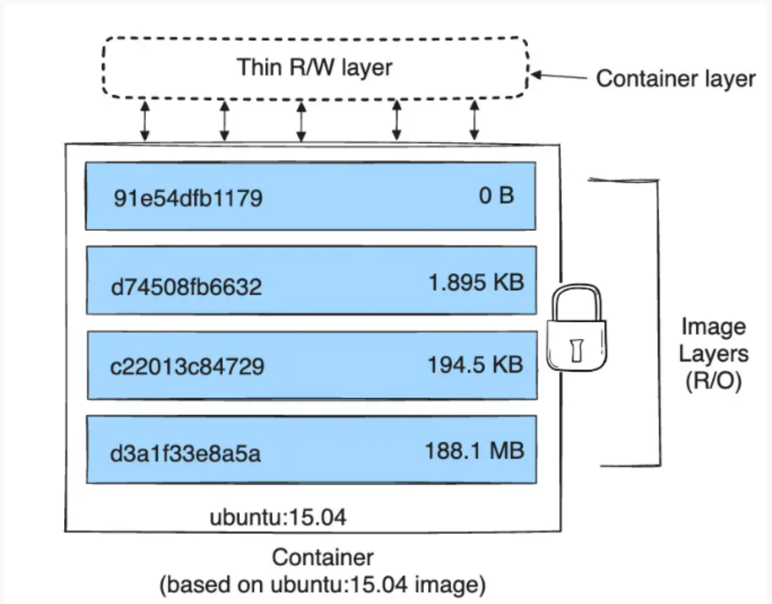
Images and layers

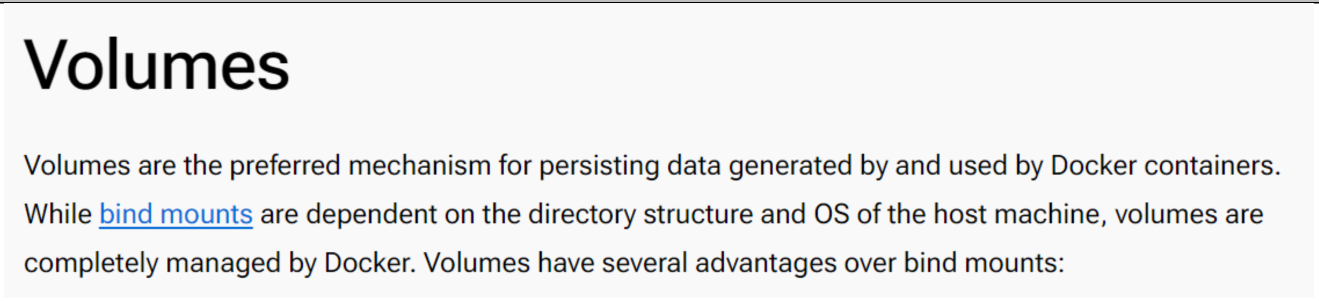
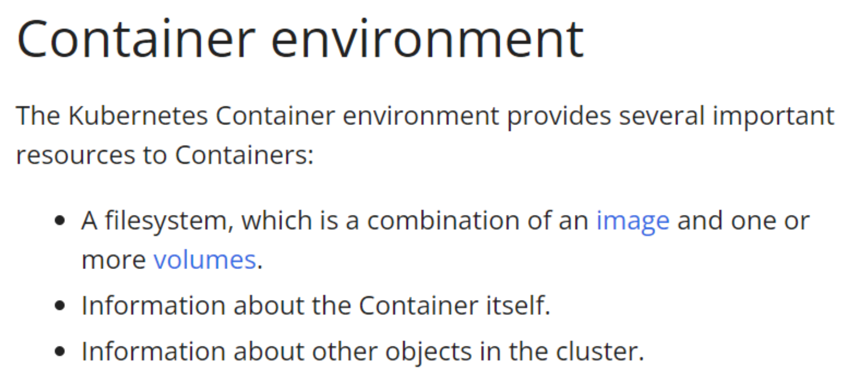
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

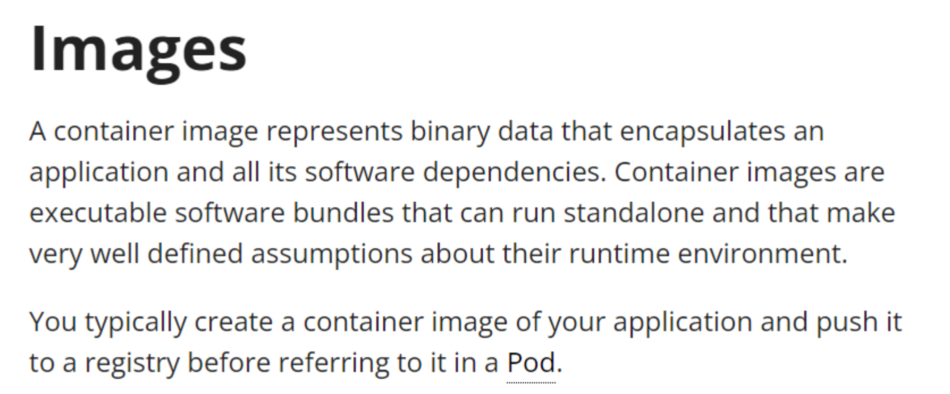
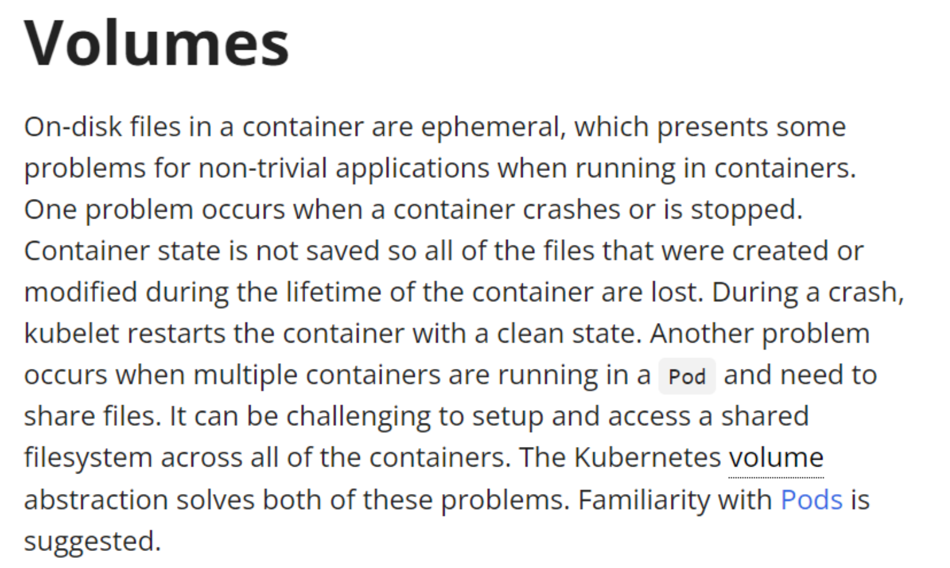
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

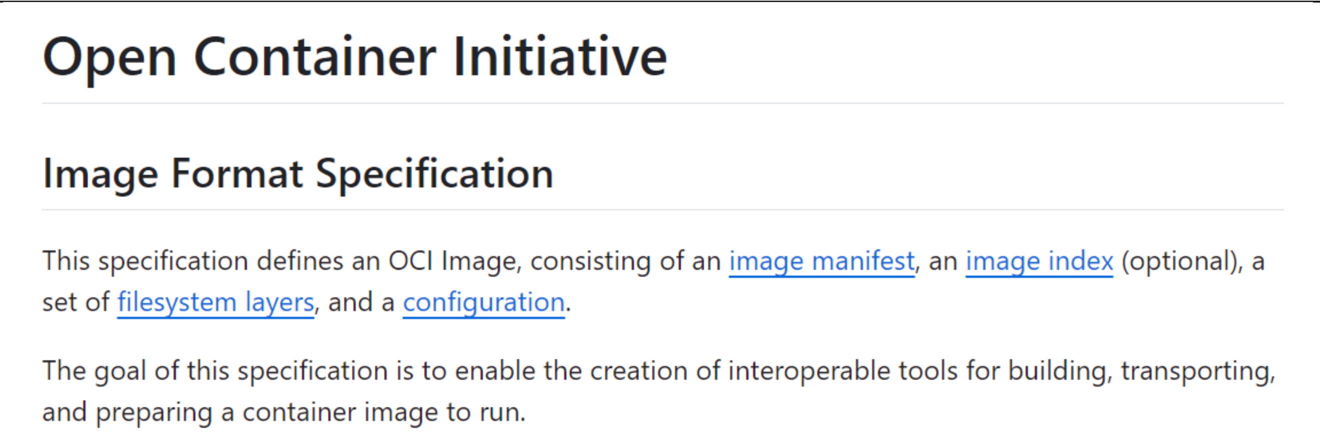
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/

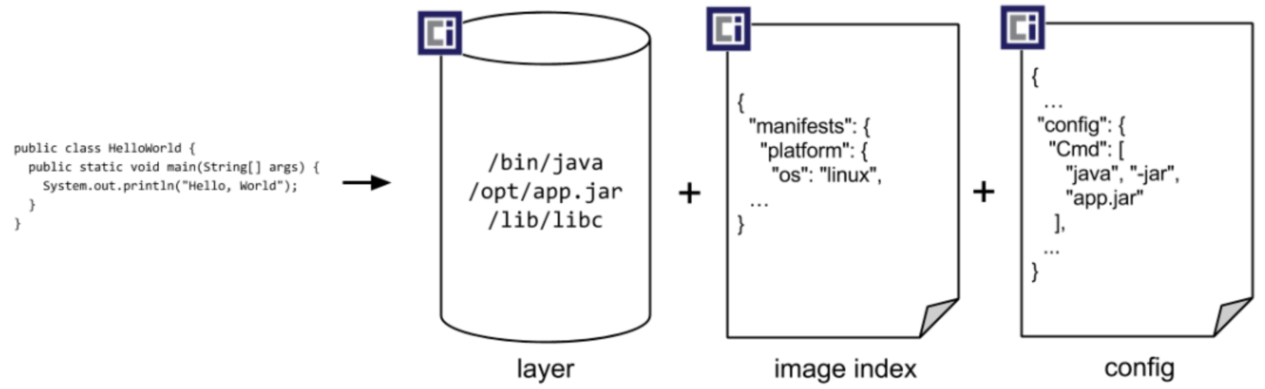
Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image Layers (R/O) Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

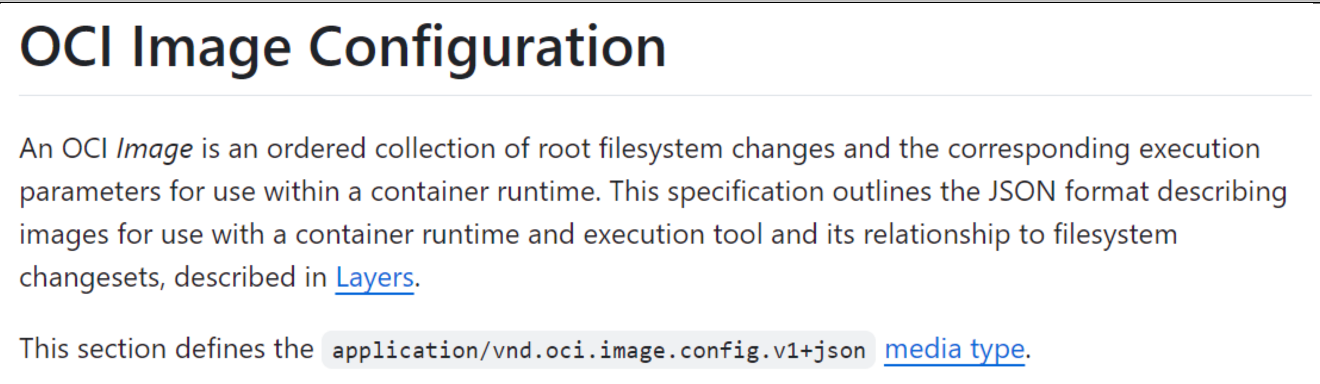
Claim 1	Accused Instrumentalities
	 **Volumes** Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/  **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

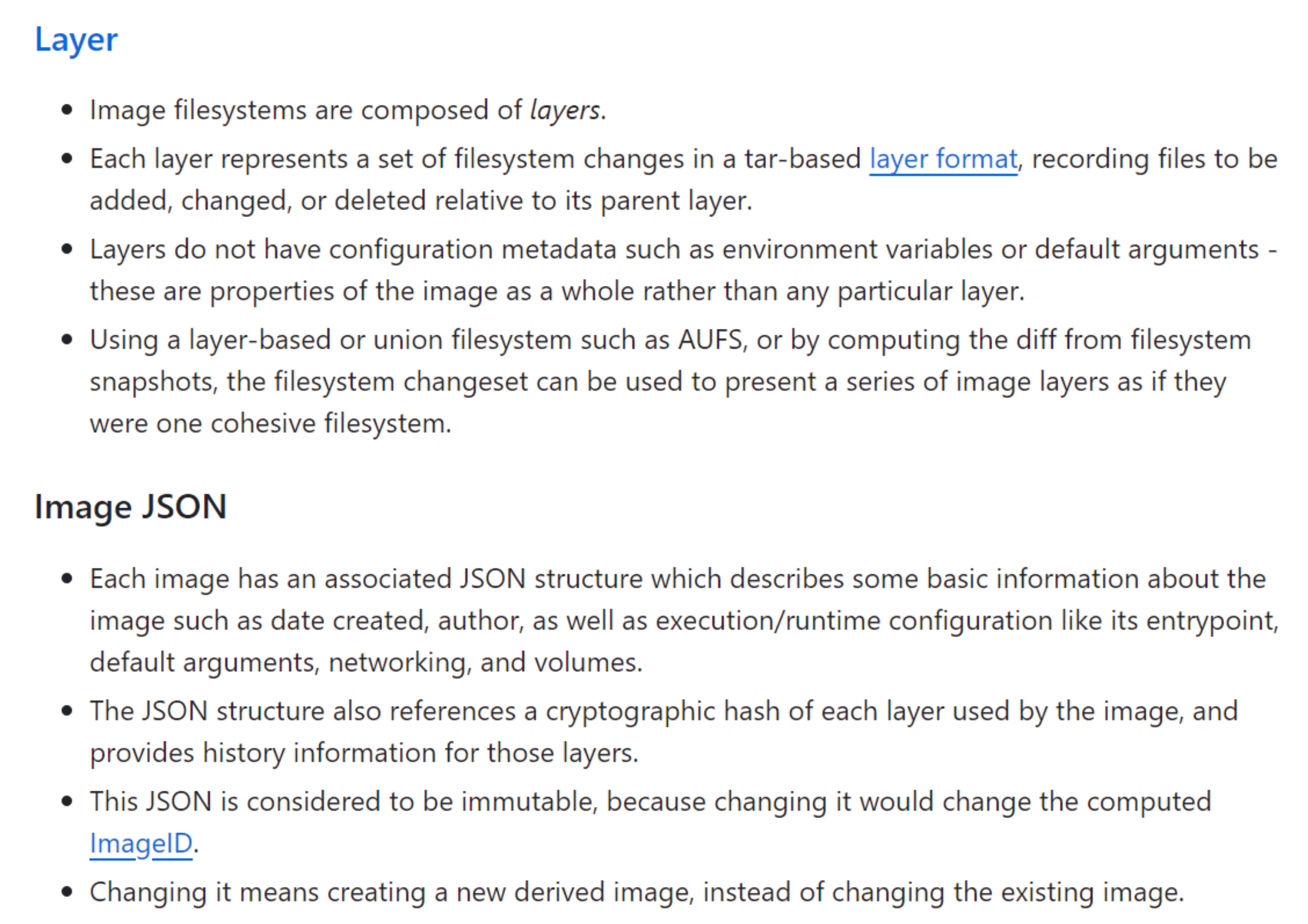
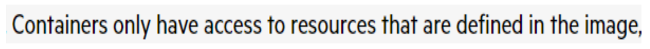
Claim 1	Accused Instrumentalities
	 **Images** A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

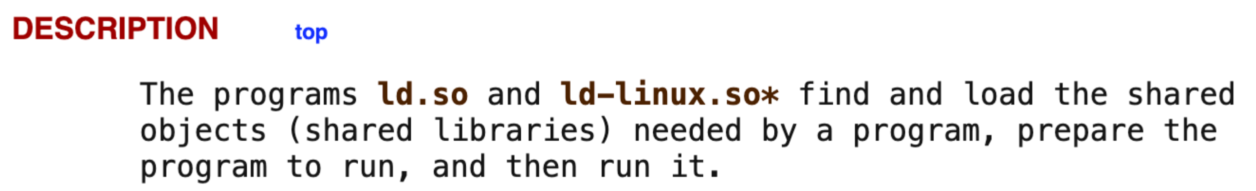
Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

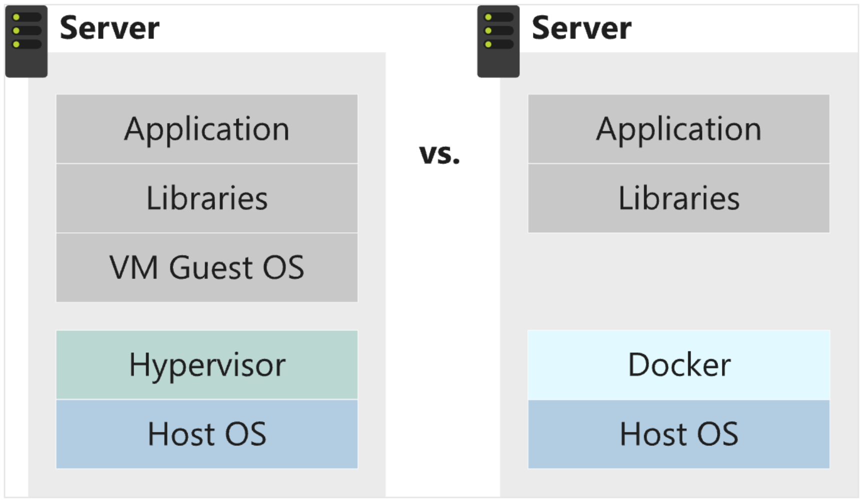
Claim 1	Accused Instrumentalities

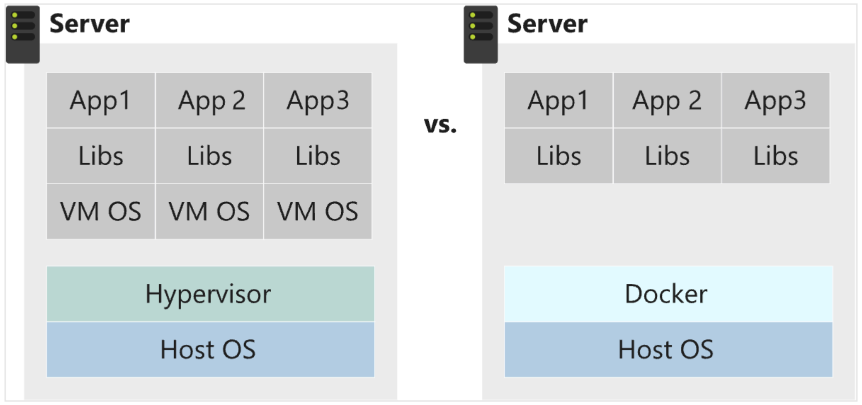
	Overview
	
	At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.
	
	
	`public class HelloWorld {`
	`  public static void main(String[] args) {`
	`    System.out.println("Hello, World");`
	`  }`
	`}`
	 /bin/java
	/opt/app.jar
	/lib/libc
	layer
	{ "manifests": { "platform": { "os": "linux", … }
	image index
	{ … "config": { "Cmd": ["java", "-jar", "app.jar"], … }
	config
	
	https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

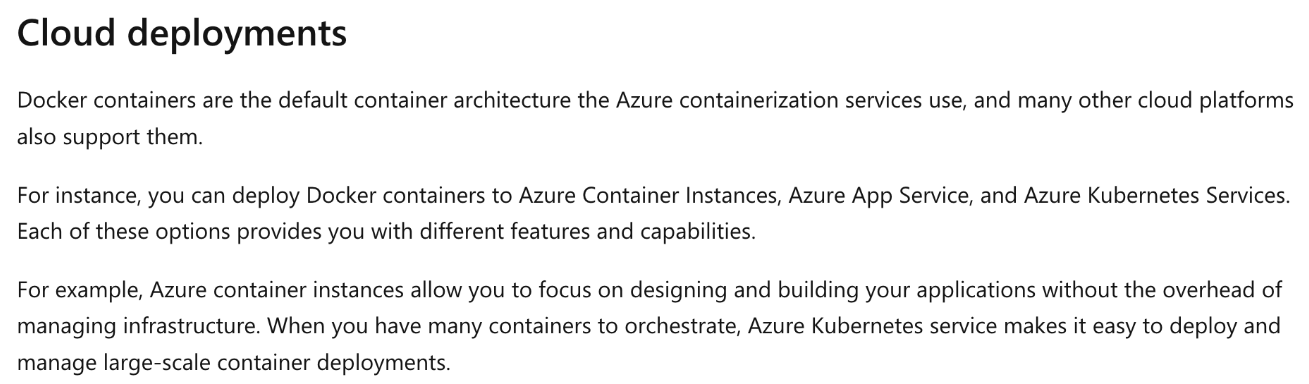
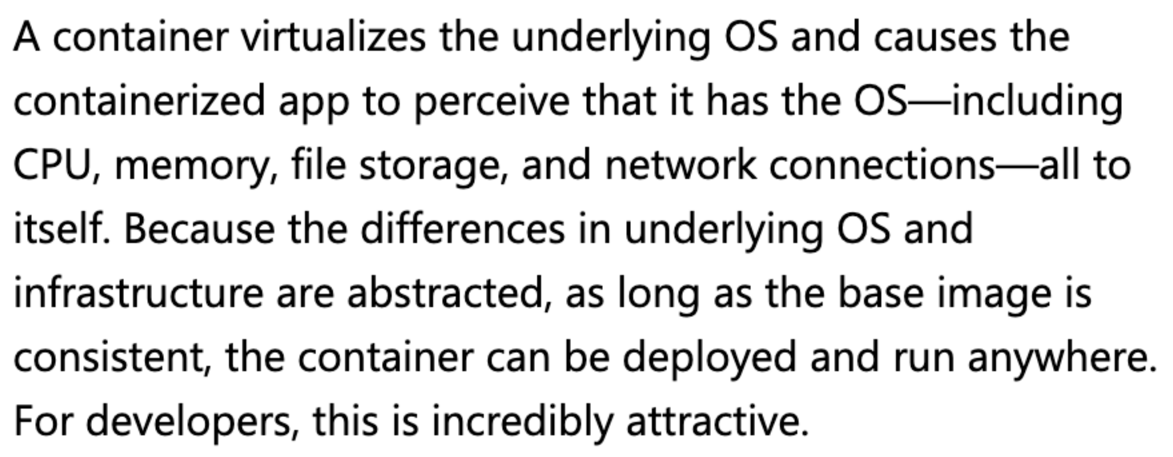
Claim 1	Accused Instrumentalities
	 OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

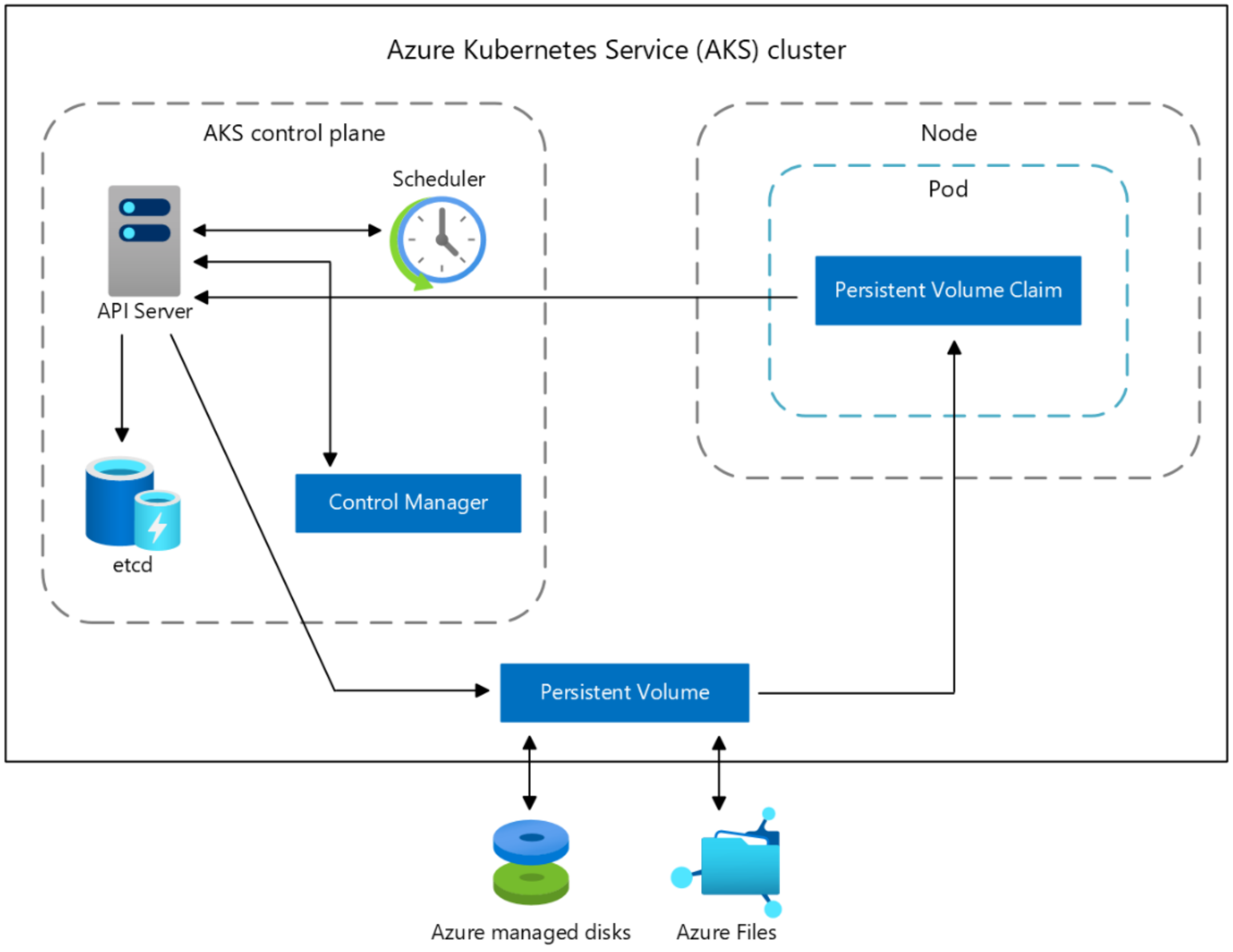
Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md  Containers only have access to resources that are defined in the image, https://www.hpe.com/us/en/what-is/docker.html

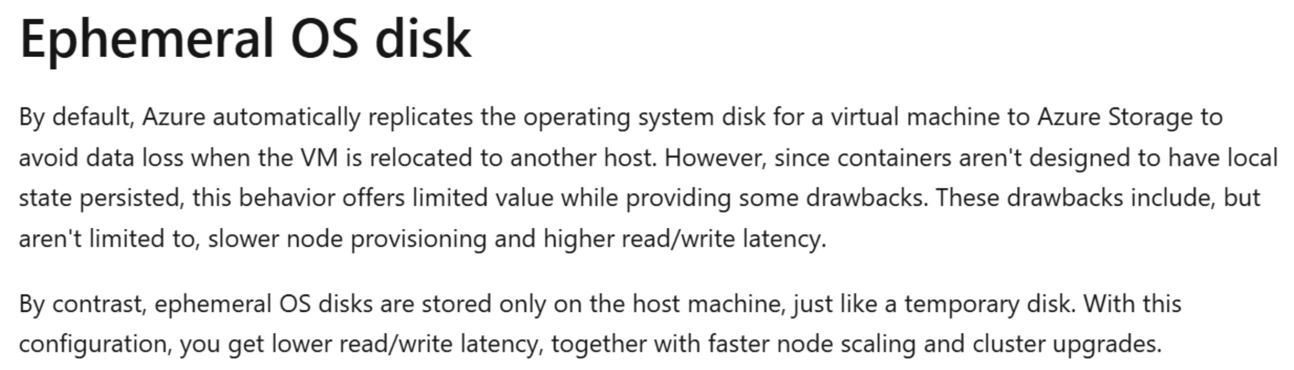
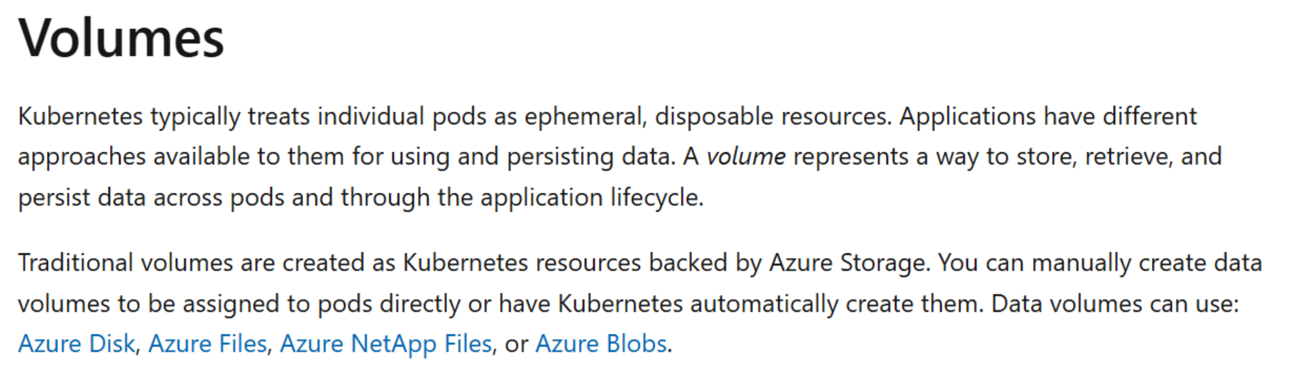
Claim 1	Accused Instrumentalities
	 **DESCRIPTION** top The programs **ld.so** and **ld-linux.so*** find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it. https://man7.org/linux/man-pages/man8/ld.so.8.html
[1e] ii) wherein an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function, and	In each Accused Instrumentality, an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function. When a Docker or Kubernetes image is used to create a container in the Accused Instrumentalities, it creates a separate and isolated instance of a runtime environment which is independent of other containers running on the same host. Each container has its own instance of base images and its own data. The containers run in isolation, ensuring that the SLCSEs stored in the shared library are accessible to the software applications running in their respective containers. The image includes essential system files, libraries, and dependencies required to run the software application within the container. The containers can share common dependencies and components using layered images. This means that multiple containers utilize the same base image to create an instance. When an instance of SLCSE is provided from the base image (i.e., from the shared library) to an individual container including application software, it is not shared with other containers. This ensures that each container has its own isolated context. Docker or Kubernetes containers in the Accused Instrumentalities can share common dependencies and components using layered images. This means that multiple containers can utilize the same base image. Therefore, each container, containing the application software running under the operating system, utilizes a unique instance of the corresponding critical system element to execute the respective application software for performing a same function. *See, e.g.*:

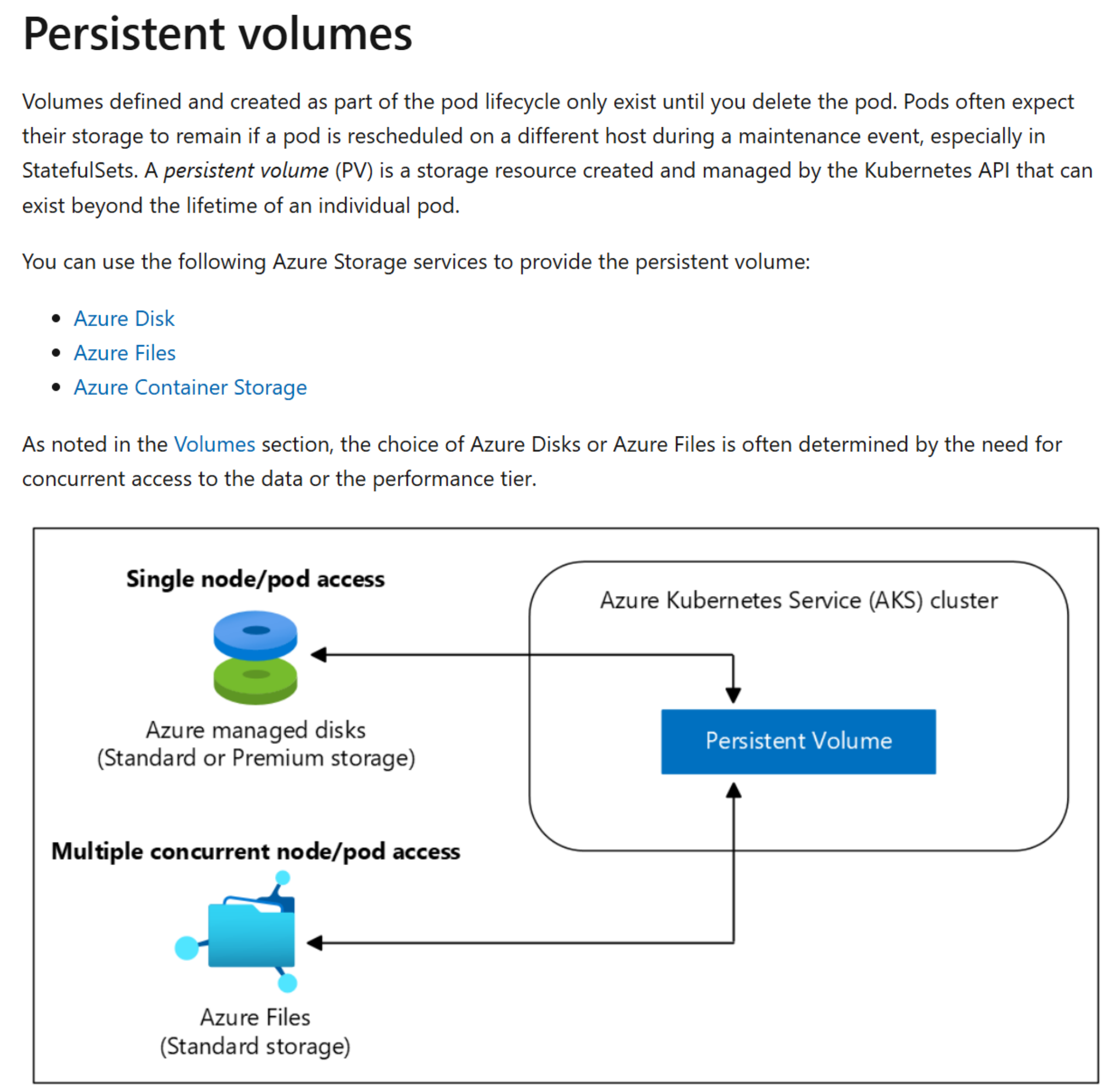
Claim 1	Accused Instrumentalities
	Docker benefits When we use Docker, we immediately get access to the benefits containerization offer. **Efficient hardware use** Containers run without using a virtual machine (VM). As we learned, the container relies on the host kernel for functions such as file system, network management, process scheduling, and memory management.  Server Application Libraries VM Guest OS Hypervisor Host OS vs. Server Application Libraries Docker Host OS Compared to a VM, we can see that a VM requires an OS installed to provide kernel functions to the running applications inside the VM. Keep in mind that the VM OS also requires disk space, memory, and CPU time. By removing the VM and the additional OS requirement, we can free resources on the host and use it for running other containers. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

Claim 1	Accused Instrumentalities
	Container isolation Docker containers provide security features to run multiple containers simultaneously on the same host without affecting each other. As we learned, we can configure both data storage and network configuration to isolate our containers or share data and connectivity between specific containers. Let's compare this feature to using VMs.  Server App1 App 2 App3 Libs Libs Libs VM OS VM OS VM OS Hypervisor Host OS vs. Server App1 App 2 App3 Libs Libs Libs Docker Host OS Assume we have a physical host running two VMs. We have three applications that we want to run isolated from each other. We decide to deploy the first app onto VM1 and the second onto VM2 to separate the two apps from each other. If we now choose to install the third application, we'll need to install another VM to continue this pattern. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers **Application portability** Containers run almost everywhere: desktops, physical servers, VMs, and in the cloud. This runtime compatibility makes it easy to move containerized applications among different environments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

Claim 1	Accused Instrumentalities
	 **Cloud deployments** Docker containers are the default container architecture the Azure containerization services use, and many other cloud platforms also support them. For instance, you can deploy Docker containers to Azure Container Instances, Azure App Service, and Azure Kubernetes Services. Each of these options provides you with different features and capabilities. For example, Azure container instances allow you to focus on designing and building your applications without the overhead of managing infrastructure. When you have many containers to orchestrate, Azure Kubernetes service makes it easy to deploy and manage large-scale container deployments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers  A container virtualizes the underlying OS and causes the containerized app to perceive that it has the OS—including CPU, memory, file storage, and network connections—all to itself. Because the differences in underlying OS and infrastructure are abstracted, as long as the base image is consistent, the container can be deployed and run anywhere. For developers, this is incredibly attractive. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-container/

Claim 1	Accused Instrumentalities
	 Azure Kubernetes Service (AKS) cluster AKS control plane Scheduler API Server etcd Control Manager Node Pod Persistent Volume Claim Persistent Volume Azure managed disks Azure Files https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	 ## Ephemeral OS disk By default, Azure automatically replicates the operating system disk for a virtual machine to Azure Storage to avoid data loss when the VM is relocated to another host. However, since containers aren't designed to have local state persisted, this behavior offers limited value while providing some drawbacks. These drawbacks include, but aren't limited to, slower node provisioning and higher read/write latency. By contrast, ephemeral OS disks are stored only on the host machine, just like a temporary disk. With this configuration, you get lower read/write latency, together with faster node scaling and cluster upgrades. https://learn.microsoft.com/en-us/azure/aks/concepts-storage  ## Volumes Kubernetes typically treats individual pods as ephemeral, disposable resources. Applications have different approaches available to them for using and persisting data. A *volume* represents a way to store, retrieve, and persist data across pods and through the application lifecycle. Traditional volumes are created as Kubernetes resources backed by Azure Storage. You can manually create data volumes to be assigned to pods directly or have Kubernetes automatically create them. Data volumes can use: Azure Disk, Azure Files, Azure NetApp Files, or Azure Blobs. https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	
	https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Persistent volumes

Volumes defined and created as part of the pod lifecycle only exist until you delete the pod. Pods often expect their storage to remain if a pod is rescheduled on a different host during a maintenance event, especially in StatefulSets. A *persistent volume* (PV) is a storage resource created and managed by the Kubernetes API that can exist beyond the lifetime of an individual pod.

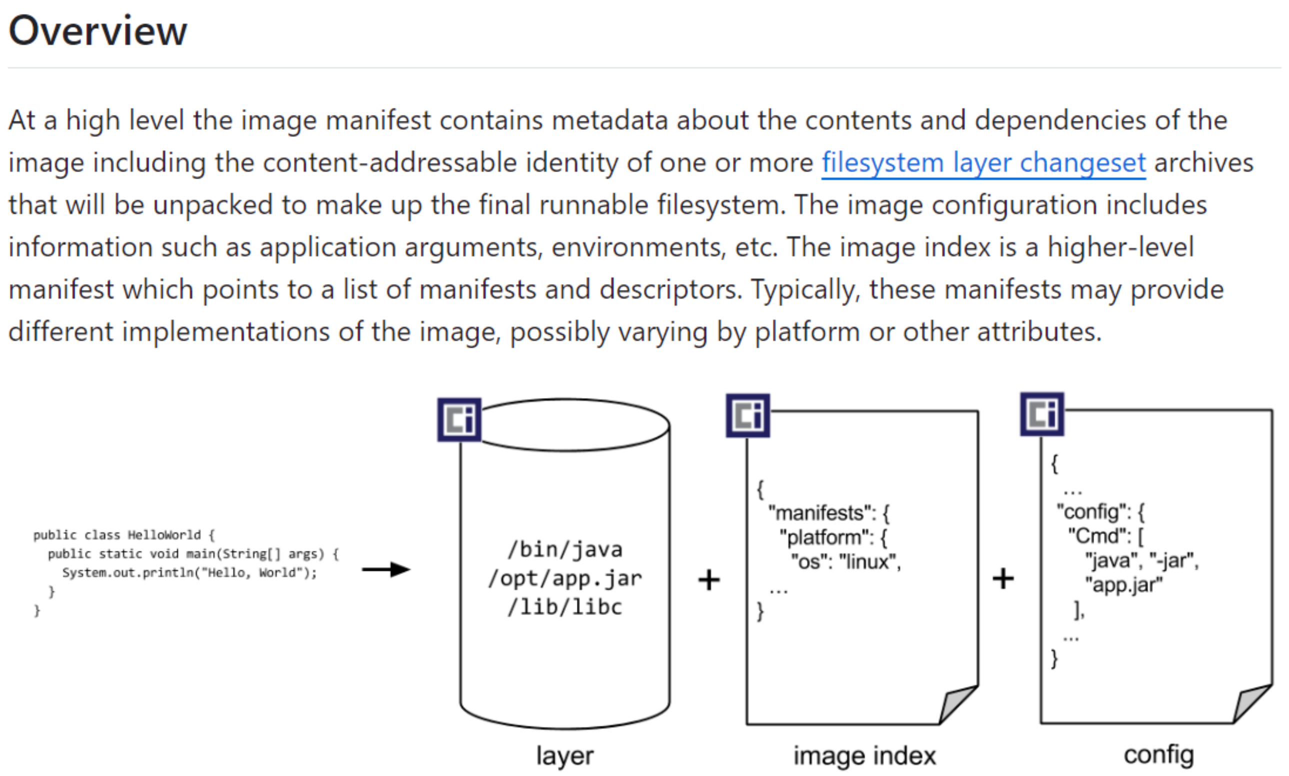
You can use the following Azure Storage services to provide the persistent volume:

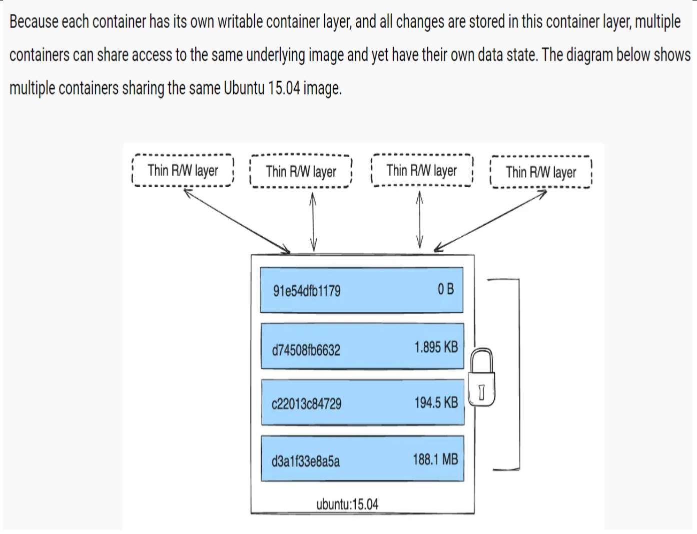
- Azure Disk
- Azure Files
- Azure Container Storage

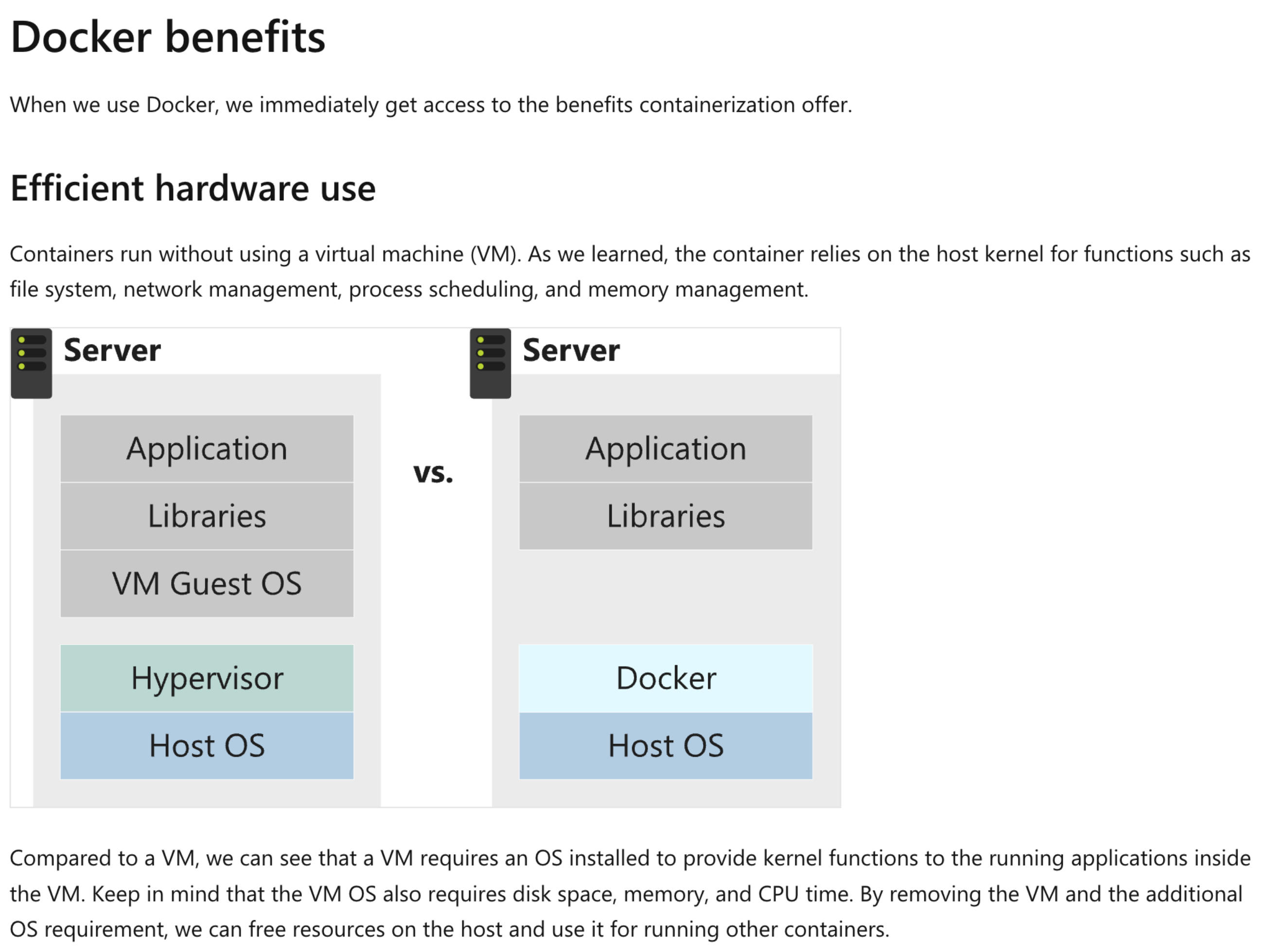
As noted in the Volumes section, the choice of Azure Disks or Azure Files is often determined by the need for concurrent access to the data or the performance tier.

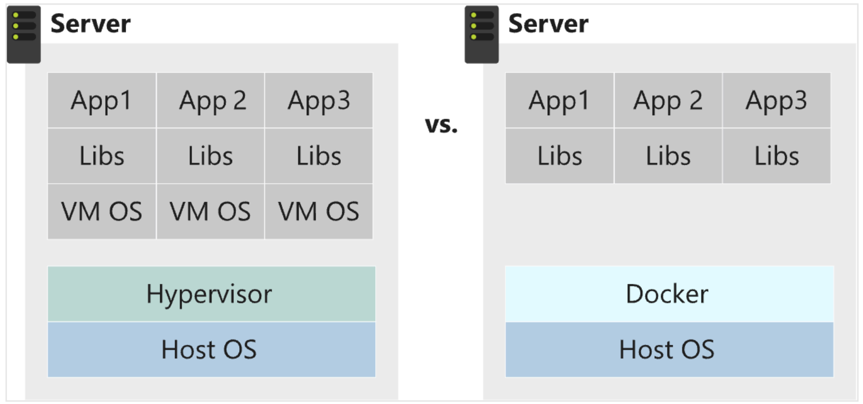
Single node/pod access
Azure managed disks
(Standard or Premium storage)
Azure Kubernetes Service (AKS) cluster
Persistent Volume
Multiple concurrent node/pod access
Azure Files
(Standard storage)

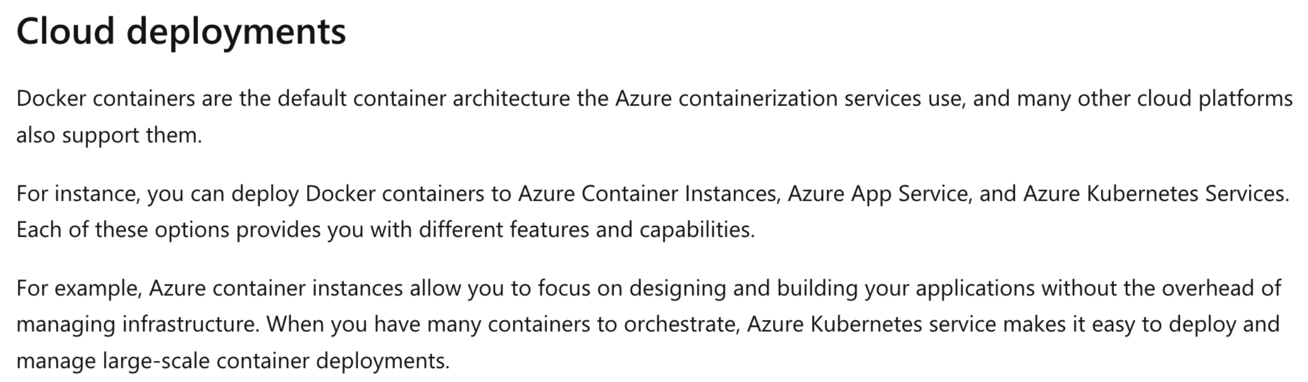
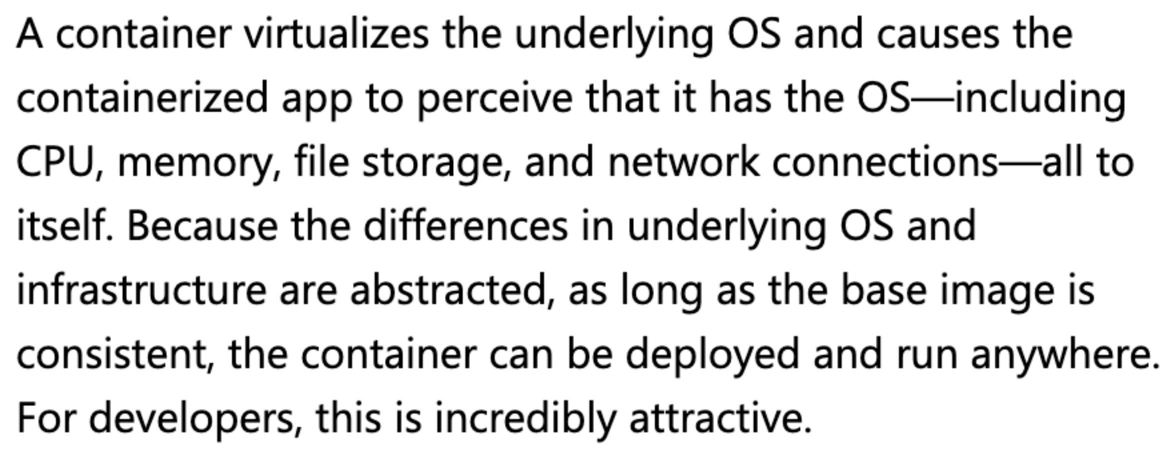
Claim 1	Accused Instrumentalities

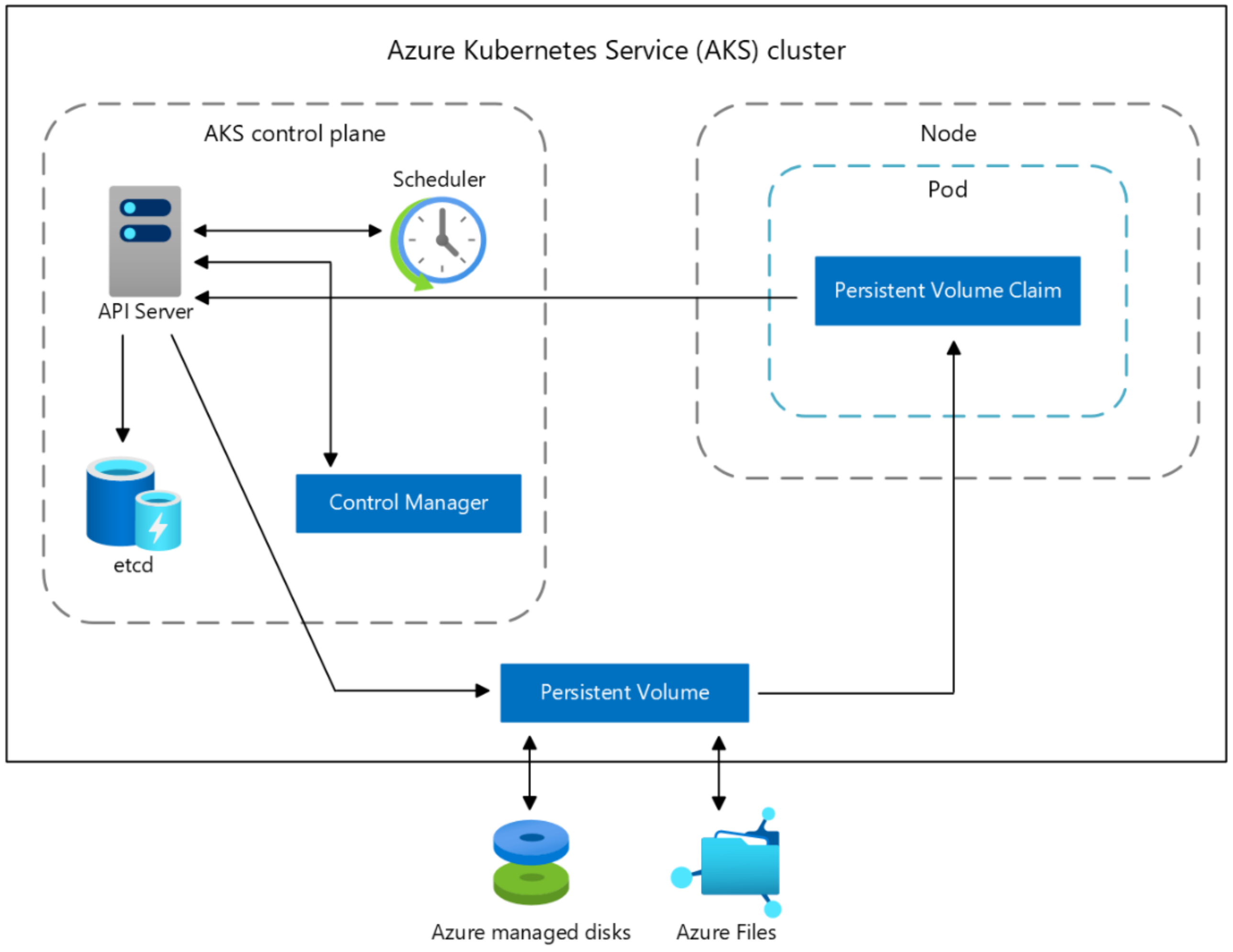
	 ## Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.
	`public class HelloWorld { public static void main(String[] args) { System.out.println("Hello, World"); } }`
	`/bin/java /opt/app.jar /lib/libc`
	`{ "manifests": { "platform": { "os": "linux", ... }`
	`{ ... "config": { "Cmd": [ "java", "-jar", "app.jar" ], ... }`
	https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

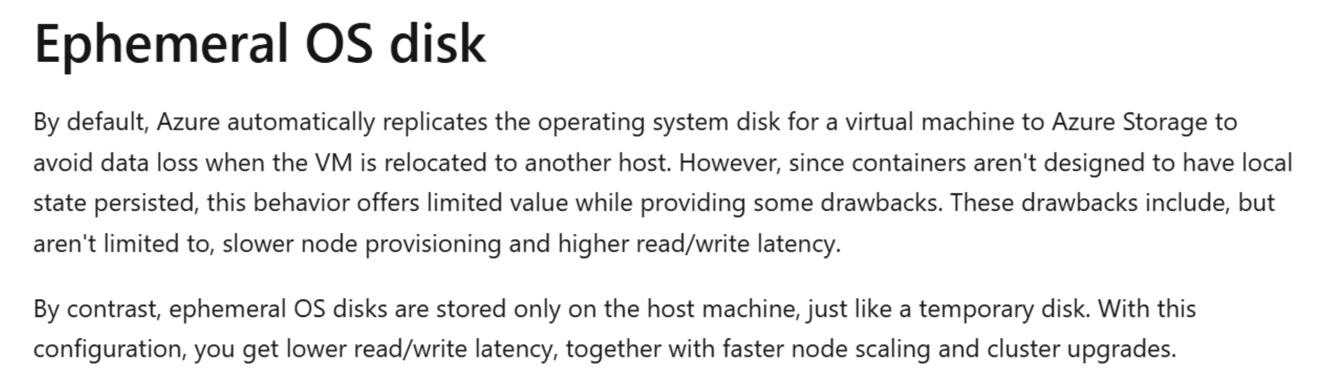
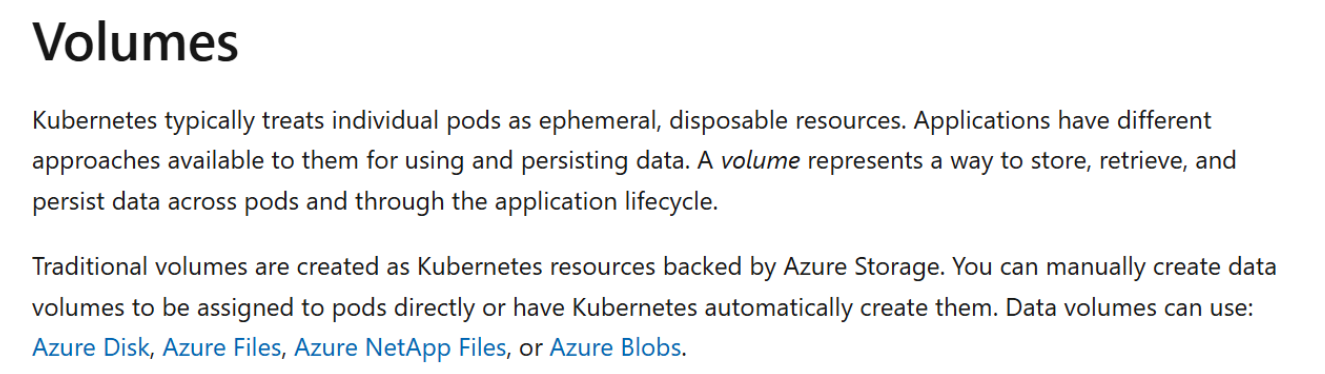
Claim 1	Accused Instrumentalities
	 Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image
[1f] iii) wherein a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously.	In each Accused Instrumentality, a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously. For example, in Docker or Kubernetes containers in the Accused Instrumentalities, each container operates independently, and a base image includes essential system files, libraries, and dependencies (i.e., SLCSEs) required to run the software application within the container. Based on information and belief, each element, such as system files, libraries, and dependencies (i.e., SLCSE) is associated with an execution of a predetermined function related to the application. When an image is used to create a container in the Accused Instrumentality, an instance of the SLCSE is provided to a software

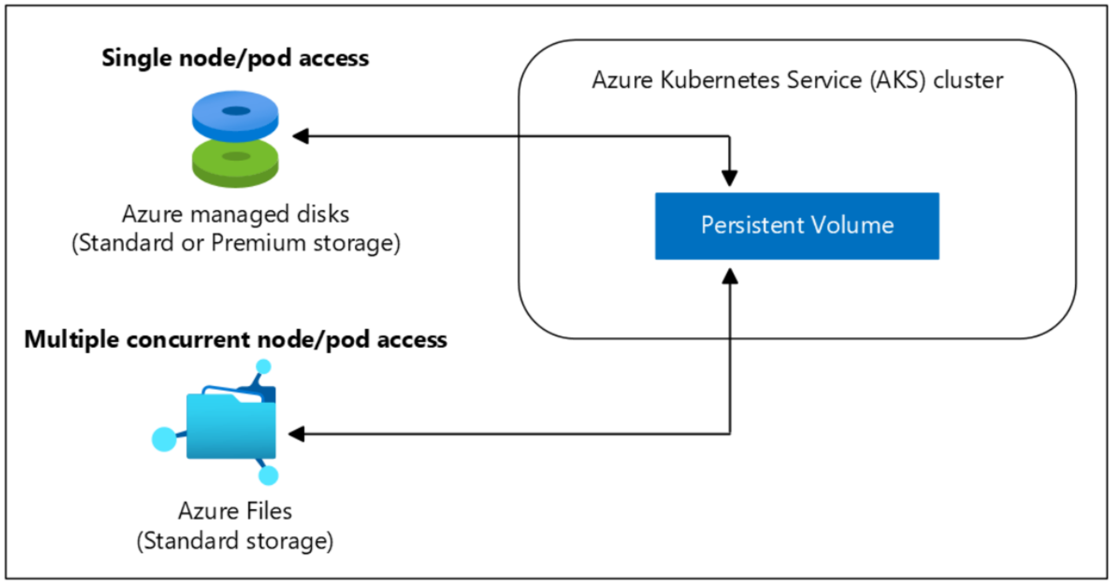
Claim 1	Accused Instrumentalities
	application. Therefore, different instances of the SLCSE are provided to different applications for performing a same function, simultaneously. *See, e.g.*:  **Docker benefits** When we use Docker, we immediately get access to the benefits containerization offer. **Efficient hardware use** Containers run without using a virtual machine (VM). As we learned, the container relies on the host kernel for functions such as file system, network management, process scheduling, and memory management. Server Application Libraries VM Guest OS Hypervisor Host OS vs. Server Application Libraries Docker Host OS Compared to a VM, we can see that a VM requires an OS installed to provide kernel functions to the running applications inside the VM. Keep in mind that the VM OS also requires disk space, memory, and CPU time. By removing the VM and the additional OS requirement, we can free resources on the host and use it for running other containers. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

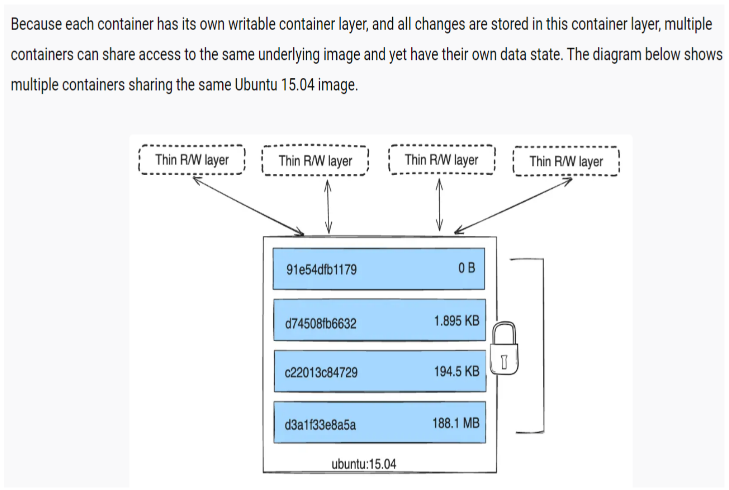
Claim 1	Accused Instrumentalities
	Container isolation Docker containers provide security features to run multiple containers simultaneously on the same host without affecting each other. As we learned, we can configure both data storage and network configuration to isolate our containers or share data and connectivity between specific containers. Let's compare this feature to using VMs.  Server App1 App 2 App3 Libs Libs Libs VM OS VM OS VM OS Hypervisor Host OS vs. Server App1 App 2 App3 Libs Libs Libs Docker Host OS Assume we have a physical host running two VMs. We have three applications that we want to run isolated from each other. We decide to deploy the first app onto VM1 and the second onto VM2 to separate the two apps from each other. If we now choose to install the third application, we'll need to install another VM to continue this pattern. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers **Application portability** Containers run almost everywhere: desktops, physical servers, VMs, and in the cloud. This runtime compatibility makes it easy to move containerized applications among different environments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers

Claim 1	Accused Instrumentalities
	 **Cloud deployments** Docker containers are the default container architecture the Azure containerization services use, and many other cloud platforms also support them. For instance, you can deploy Docker containers to Azure Container Instances, Azure App Service, and Azure Kubernetes Services. Each of these options provides you with different features and capabilities. For example, Azure container instances allow you to focus on designing and building your applications without the overhead of managing infrastructure. When you have many containers to orchestrate, Azure Kubernetes service makes it easy to deploy and manage large-scale container deployments. https://learn.microsoft.com/en-us/training/modules/intro-to-docker-containers/5-when-use-docker-containers  A container virtualizes the underlying OS and causes the containerized app to perceive that it has the OS—including CPU, memory, file storage, and network connections—all to itself. Because the differences in underlying OS and infrastructure are abstracted, as long as the base image is consistent, the container can be deployed and run anywhere. For developers, this is incredibly attractive. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-container/

Claim 1	Accused Instrumentalities
	 Azure Kubernetes Service (AKS) cluster AKS control plane Scheduler API Server etcd Control Manager Node Pod Persistent Volume Claim Persistent Volume Azure managed disks Azure Files https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	 Ephemeral OS disk By default, Azure automatically replicates the operating system disk for a virtual machine to Azure Storage to avoid data loss when the VM is relocated to another host. However, since containers aren't designed to have local state persisted, this behavior offers limited value while providing some drawbacks. These drawbacks include, but aren't limited to, slower node provisioning and higher read/write latency. By contrast, ephemeral OS disks are stored only on the host machine, just like a temporary disk. With this configuration, you get lower read/write latency, together with faster node scaling and cluster upgrades. https://learn.microsoft.com/en-us/azure/aks/concepts-storage  Volumes Kubernetes typically treats individual pods as ephemeral, disposable resources. Applications have different approaches available to them for using and persisting data. A *volume* represents a way to store, retrieve, and persist data across pods and through the application lifecycle. Traditional volumes are created as Kubernetes resources backed by Azure Storage. You can manually create data volumes to be assigned to pods directly or have Kubernetes automatically create them. Data volumes can use: Azure Disk, Azure Files, Azure NetApp Files, or Azure Blobs. https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	Persistent volumes Volumes defined and created as part of the pod lifecycle only exist until you delete the pod. Pods often expect their storage to remain if a pod is rescheduled on a different host during a maintenance event, especially in StatefulSets. A *persistent volume* (PV) is a storage resource created and managed by the Kubernetes API that can exist beyond the lifetime of an individual pod. You can use the following Azure Storage services to provide the persistent volume: • Azure Disk • Azure Files • Azure Container Storage As noted in the Volumes section, the choice of Azure Disks or Azure Files is often determined by the need for concurrent access to the data or the performance tier.  Single node/pod access Azure managed disks (Standard or Premium storage) Azure Kubernetes Service (AKS) cluster Persistent Volume Multiple concurrent node/pod access Azure Files (Standard storage) https://learn.microsoft.com/en-us/azure/aks/concepts-storage

Claim 1	Accused Instrumentalities
	Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state. https://docs.docker.com/get-started/overview/  Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image. Thin R/W layer Thin R/W layer Thin R/W layer Thin R/W layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 https://docs.docker.com/storage/storagedriver/